MINIMIZATION OF SUPERVISOR CONFLICT FOR MULTIPROCESSOR COMPUTER SYSTEMS

A THESIS

Presented to

The Faculty of the Division of Graduate

Studies and Research

by

Randy J. Raynor

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

in the School of Information and Computer Science

Georgia Institute of Technology

June, 1974

MINIMIZATION OF SUPERVISOR CONFLICT

FOR MULTIPROCESSOR COMPUTER SYSTEMS

Approved:

John M. Gwynn, Jr., Chairman

Michael D. Kelly

Donovan B. Young

Date approved by Chairman: 5/25/74

ACKNOWLEDGMENTS

I would first like to thank my thesis advisor, Dr. John Gwynn, for his support, ideas, and constructive criticism throughout my graduate career. Our numerous discussions have been of inestimable value to me.

The members of my guidance and reading committees, Drs. Robert Cooper, Michael Kelly, and Donovan Young deserve my thanks for their continued support and encouragement during my research and for their careful reading of my thesis drafts. I would also like to thank Dr. Donald Chand of Georgia State University and Dr. James Browne of The University of Texas at Austin for reading and commenting on this thesis.

A note of thanks goes to Mr. Bill Brown of Univac and to the Office of Computing Services of Georgia Tech for their assistance in setting up the special use of the Univac computer. This research was partially supported by NSF Grant GN-655.

Finally, special thanks are given to my wife, Vickie. Her assistance, patience, and impatience were an essential contribution to the completion of this thesis.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF TABLES

LIST OF ILLUSTRATIONS

SUMMARY

An investigation is made into a basic design problem of multiprocessor computer systems resulting from queueing of requests for the supervisor. For this study, a computer system simulator is constructed which represented a portion of the structure of a generalized multiprocessor system. A methodology is developed to enable the scheduling of tasks to processors such that subsequent queueing of requests for the supervisor would be reduced, thereby increasing throughput.

The capability to accomplish this is based on an assumed knowledge of the exact processing and I/O requirements of the tasks in the system's workload. A general analysis studies the effects of inaccuracies in this knowledge on the methodology. Then an estimate of the expected degree of accuracy of this information is determined by applying several forecasting techniques to task-characteristic data obtained through software monitoring of actual programs.

A set of experiments is performed and statistically analyzed which compares the methodology developed here with conventional techniques using system throughput as the basic measure of improvement obtained. Results indicate that significant improvement could be obtained, e.g., throughput is increased by seven per cent for a 21 processor system under specified conditions.

CHAPTER I

INTRODUCTION

1.1 Goals of This Research

1.1.1 Description of the Problem

In a multiprocessor system, the handling of interrupts generated by jobs in the processors is assigned to a supervisory program and associated data base. The two basic philosophies for deciding which processor executes the supervisor are master-slave and floating executive control (78). In either case, queueing of requests to the supervisor may occur. With the master-slave structure, the master processor can handle only one request at a time. With floating executive control, while any processor can execute the supervisor, only one processor can be allowed to access the supervisor's data base at a time.

A processor which is waiting to use the supervisor is not doing useful work. Therefore, queueing of requests to the supervisor causes a degradation in the performance of the system. Previous studies have indicated that this degradation is significant for large multiprocessor systems (66).

1.1.2 Summary of the Methodology

Jobs in the system's workload are characterized by their processing and I/O requirements. Supervisor queueing will be reduced by using this information to schedule jobs to processors such that they request the use of the supervisor when the supervisor is predicted to be available. This methodology will be developed and evaluated via a computer

system simulator which is of a sufficient level of detail to model the flow of tasks through the system's major resources.

The capability to accomplish this is based on an assumed knowledge of the exact processing and I/O requirements of the tasks in the system's workload. A general analysis will study the effects of inaccuracies in this knowledge on the methodology. Then a realistic estimate of the expected degree of accuracy of this information will be determined by applying several forecasting techniques to task-characteristic data obtained through software monitoring of actual programs.

A set of experiments will be performed comparing the methodology developed here with conventional techniques, using system throughput as the basic measure of the improvement obtained. Since the experiments are essentially Monte Carlo in nature, sufficient care will be taken in their design to assure reasonable statistical confidence.

1.2 Summary of Following Chapters

Chapter II provides an extensive survey of multiprocessor computer systems, problems inherent in these systems, and techniques which are used to solve these problems. Included is the concept of system design which allows the system to react dynamically to the workload in order to improve performance.

Chapter III develops the methodology for the reduction of supervisor queueing. The simulation models used to develop the scheduler are described, and expected results are intuitively described.

The results of the simulation studies are given in Chapter IV, and Chapter V provides possible extensions of the methodology to other areas of application and summaries with conclusions about the feasibil-

ity of the methodology.

1.3 Note on Terminology

This thesis assumes a basic understanding of computer science concepts and uses standard computer science terminology without including definitions except where deemed necessary. The reader is referred to standard computer science texts (6,96) for those terms with which he is not familiar.

CHAPTER II

MULTIPROCESSOR SYSTEM DESIGN

A multiprocessor computer system is one which has more than one central processing unit. Multiprocessing should not be confused with multiprogramming which refers to the interleaved execution of many programs which are, themselves, executed sequentially (96). Multiprogramming makes no reference to the number of processors involved in this execution.

2.1 Design Alternatives

2.2.2 Structural Alternatives

There are many different types of multiprocessing systems; Flynn has developed a scheme for catagorizing them (40). This scheme is based on the notion of streams. The four classifications arise from the multiplicity of two types of streams: streams of instructions, and streams of data. Flynn's SISD classification includes those systems which are based on a single instruction stream (SI) and a single data stream (SD). Most of the current single processor systems fall into this category. Computers which have a single instruction stream and a multiple data stream (SIMD) include array processors like the Illiac IV, pipelined processors like the Solomon, and associative processors (40). The multiple instruction stream, single data stream organization (MISD) is not practical today. The last category, multiple instruction stream, multiple data stream (MIMD) has two variants. The first is called a

Shared Resource Multiprocessor. This includes systems which have "skeleton" or incomplete processors which share system resources as in the CDC 6600 (6). The second is called a True Multiprocessor System. Here several physically complete and independent SI processors execute separate tasks or subtasks.

Another separate classification of multiprocessor systems uses more of the current terminology in its specification of six categories (7). The first category, parallel processing, involves multiple processors which are assigned to independent subtasks of a task which can be processed simultaneously. The major problem with this type of processing is in deciding how to break a task into subtasks. The second category, pipeline processing, has several different arithmetic processing units which perform one part of a single operation and then pass the result on to the next "processor" until the entire operation is completed. With this structure, the same operation can be at various degrees of completion on different data. Recall that Flynn classified this as SIMD. The third category, network processing, involves computers which have special function subsystems which may themselves be multiprocessor systems. This is similar to Flynn's MIMD Shared Resource Structure. The fourth category includes multiprocessor systems which have specialized hardware processors to perform functions like interpreting. The fifth category uses conventional multiprogramming on systems with more than one processor. This is like Flynn's MIMD True Multiprocessors. The last category, called Independent Computing, has subsystems hardware partitioned into distinct entities.

2.1.2 Justification for Multiple Processors

This research will involve one specific category: True Multiprocessor Systems. Before going into some of the details of the design of this type of system, some justifications will be given for multiprocessor systems in general, and True Multiprocessor Systems in particular (98). Hereafter, True Multiprocessor Systems will be referred to simply as multiprocessor systems.

The first obvious advantage of having multiple processing units is the increase in throughput, even though it may not be true that a two processor system can do twice as much work as one processor. A system which has multiple processing units will usually have multiple channels, multiple card readers, multiple disk subsystems etc. This multiplicity provides both increased efficiency and increased reliability. Multiple resources are more efficient when they service requests from the same queue rather than from different queues because this structure prevents one resource from being idle while a queue exists at a similar resource. Increased reliability results from redundant components if the system is designed such that failing components can be dynamically configured out of the system. This capability is usually referred to as graceful degradation. Also, large multiprocessing systems can accommodate large programs which would require more resources than are usually available on single processor systems. Another advantage which one large system has over many small ones is that memory would be more effectively utilized because only one copy of large data files, compilers, operating system, etc., would be needed.

The previous discussion has indicated the advantages of multi-

processor systems, but there is one important factor which makes these systems feasible: The cost of processing units has been reduced much faster than the cost of some other system resources (7).

Based on justifications such as these, it has been suggested that a trend toward multiprocessor systems is expected in the near future (7). In 1968 Witt explained that it was at that time undetermined how well multiprocessor systems could meet expectations (98). There are currently many design problems which must be considered before their effectiveness is determined. Some of the design considerations for multiprocessor systems will be discussed in the next section.

2.1.3 Special Considerations for True Multiprocessor Systems

When a multiprocessor system is under design, there are many design problems which must be considered and then resolved. One such problem is the technique for maintaining control over the processors. Since the supervisor is the control mechanism, the question which must be resolved is: Which processor(s) will be allowed to execute the supervisor?

There are two basic structual alternatives. First, there is the master-slave control (6). In this situation, one specific processor, the master, is the one and only one allowed to execute the supervisor. The other processors, the slaves, execute only problem programs. In some cases, the master is also allowed to execute problem programs. This structure implies that if the job in a slave processor causes an interrupt, that slave must wait for the master to handle that interrupt. If the master was already handling a previous interrupt, then a queue of requests for the supervisor will develop. This structure eliminates

one of the advantages of multiprocessor system--graceful degradation. If the master processor fails, then the entire system must stop.

The second design alternative, floating supervisor control, does provide for graceful degradation. In this situation, the supervisor is considered a resource which any processor can request. Thus, when a job in a processor generates an interrupt, that processor can request the use of the supervisor to handle that interrupt. If there were only one non-reentrant copy of the supervisor, then only one processor could use the supervisor at a time. If there were more than one copy, or if it were reentrant, then more than one processor could possibly be using the supervisor at the same time.

However, there would have to be limits placed on the simultaneous use of the supervisor. A "critical race" would occur if one processor were trying to change the supervisor's data base while another processor was trying to access that data base (14). For example, if two processors were using the job scheduling algorithm at the same time, they could both select the same job to execute.

The usual technique for dealing with this problem is the use of a LOCK-UNLOCK flag (14,33). When one processor wanted to use the supervisor, it would LOCK other processors from having access to it. Upon completion, it would UNLOCK the supervisor. Note that this could be applied to the supervisor as a whole, or separate locks could be used for separate parts of the supervisor which access distinct parts of the supervisor's data base. The use of locks could become expensive in terms of both time and complexity of the supervisor. Some examples of these control structures will be given in a later section.

There are other structual questions besides control which must be resolved in the design of a multiprocessor system. The current design of core memory would prevent the access of memory by more than one processor at a time. The first step in alleviating this situation is to organize central memory into blocks of memory, such that different blocks can be accessed simultaneously by different processors. But having more than one memory block introduces the problem of how the processors will be connected to the memory blocks.

Critchlow has described three current techniques for implementing this connection (32). One would be a time-shared bus in which processors and memory blocks are connected only long enough for the transfer, and then the connector is switched to satisfy another processor-memory transaction. A slightly more expensive and somewhat faster technique would be to hardware connect each processor to several particular memory blocks; thereby allowing them to access only those blocks. The fastest, most general, and most expensive technique is called a crossbar switch. Here every processor is connected to every memory block.

Based on a simulation model of a multiprocessor system with a crossbar switch, Lehman studied the problem of specifying the number of memory blocks necessary to support a given number of processors (6). His results indicate that a memory/processor ratio of 4/1 may be necessary. An extensive study of this problem through analytic models has been made by Bhandarkar (8).

Another design problem which must be considered involves the decision as to which processor will be allocated to handle external interrupts such as an I/O complete interrupt. Some of the alternatives

include: 1) the processor which initiated the I/O activity, 2) a designated processor which handles all such interrupts 3) a processor selected by some algorithm. Goutanis has developed a sophisticated scheme which fits into this last category (46).

In summary, three problem areas in multiprocessor system design have been discussed: one which deals with communication among processors; one which deals with communication between processors and memory; and one which deals with communication between processors and channels. Before proceeding with a detailed investigation of one of these areas, some examples of existing multiprocessor designs will be given.

2.2 Survey of Existing Systems

There are a large number of multiprocessor computers currently in operation or under design. Most computer manufacturers have mainline products based on a multiprocessor structure, and there are many installations which have built multiprocessor systems out of single processor systems. Several examples of these systems will now be discussed (84). First, they will be described by specifying the maximum number of major components to indicate the degree of multiplicity. Second, the basic characteristics of the operating system, such as the scheduling algorithm, will be mentioned as an indication of the degree of sophistication of design. Also, any special features important to this research will be mentioned.

2.2.1 Mainline Products

The Burroughs 700 series of computers includes several which have multiprocessing capabilities: the B5700, the B6700, and the B7700. The

B5700 allows up to two processors and four channels. One of these processors is designated to execute the supervisor and problem programs, and the other, only problem programs. The B6700 allows up to three processors and 12 channels while the B7700 can accommodate eight processors, 32 channels, and eight independent memory modules. Both of these systems are based on the floating supervisory control scheme.

The operating system, MCP, supports batch, real-time, and time-sharing operations. The Burroughs virtual memory scheme is based on program segmentation. Job scheduling uses a dynamic priority assignment, but with provisions which allow adjustments in the mix to balance the load on the machine by keeping as much of the system as possible busy at the same time.

The Univac 1110 (1108) will support up to six (four) processors and 96 (64) channels. All processors are allowed to execute both supervisory and problem programs.

The EXEC operating system schedules jobs from its virtually unlimited mix according to a priority scheme. The scheduler also has the capability to take into consideration job deadlines.

The Honeywell 615/625/635 and the newer 6000 are multiprocessor machines. The maximum number of processors on the 615 is three, and the 6000 can support up to four.

The GECOS III operating system is advanced in design and scope. Incorporated in GECOS III are extensive techniques for event tracing and utilization monitoring (20). Jobs are scheduled from a maximum mix of 63 according to a timesliced round robin priority scheme. The 600 series utilizes a master-slave control philosophy.

The design of the CDC 6400/6500/6600/7600 multiprocessor systems is quite different from those systems previously described. Basically these machines have either one or two central processors and seven to twenty peripheral processors. Each peripheral has 4K of memory and there are usually 32 independent 4K banks of central memory which provide a connection between the central processor and the peripheral processors. The peripheral processors control input/output functions and provide work for the central processor, which is an extremely fast arithmetic unit. The 7600 can have up to 24 I/O channels.

One of the peripheral processors is designated to execute the operating system, and thereby act as the master processor for the system. The operating system, SCOPE 3, maintains timesharing and local and remote batch facilities. Job scheduling is accomplished through an extensive priority system.

The CDC machines are considered to be the biggest and fastest of the commercially available computers. They are often used in scientific environments, where such power is effectively utilized (6,28,84).

2.2.2 Special Products

Since 1968 the IBM facility at Gaithersburg, Maryland, has had a system designated M65MP which is composed of two model 65 computers connected both directly and through common main storage (98). The model 65 operating system was only slightly modified to accommodate this structure. Basically, the resource allocation was modified so that peripherals of eigher CPU could be used by either CPU. The various components of the supervisor were classified into two groups. One group contained those routines which would be affected if both processors

tried to execute them simultaneously. A simple lock was provided to prevent such simultaneous access. The other group contained those routines which would not be affected by simultaneous execution.

Prior to the M65MP, another dual processor system was developed from combining an IBM 709 with a 704 (58). In this system the 704 acted as the master processor and was not allowed to execute problem programs.

Another IBM multiprocessor system, 9020, was designed to handle real-time air route traffic control (33,67). This system could incorporate up to four mid-range 360 CPU's, nine channels, and twelve independent main storage elements. A floating supervisor control scheme was used which locked out parts of the supervisor which accessed a common data base to prevent the race condition. The scheduling algorithm was extended beyond the usual capabilities to allow the dynamic rescheduling of an interrupted job while another processor handled the interrupt generated by the job. This permitted the shortest possible elapsed times for critical real-time jobs.

Burroughs developed a multiprocessing system with real-time capabilities similar to that in the IBM 9020 called the D825 (6). The maximum system configuration consisted of four processors, 16 4K memory modules, and 20 channels. The AOSP operating system was designed to allow floating supervisor control. The scheduling algorithm was a dynamic priority scheme which insured maintenance of specified procedure relations among jobs.

NASA in Houston has developed a multiprocessor system consisting of two UNIVAC 1106's, four UNIVAC 1108's, and a large complement of peripherals (76). Northouse has developed a load-balancing scheduling

algorithm for this system which will be discussed in some detail later (76).

The Hughes Aircraft Company has designed the H-3118 multiprocessor system consisting of three processors and eight 16K memory banks (78). A single LOCK was used to refer to the entire supervisor so that only one processor could access it at a time.

Carnegie Mellon University is in the process of designing and constructing a multiprocessor system based on up to 16 miniprocessors, in particular, 16 PDP 11's. Supporting research has investigated in detail some of the problems in multiprocessor systems as previously described. Analytic models of the memory interference problem have been developed (8), and studies of the structure of the operating system have begun (99). The operating system will be designed around a kernel which contains the basic mechanisms for building an operating system but no specific policies such as scheduling philosophies.

2.3 Process of Multiprocessor Design

Thus far, various justifications for multiprocessor systems, various design philosophies, and various examples have been given. Also, one effect of the current technology on the feasibility of the systems, i.e., the reduction in processor cost, has been introduced. Now a very important area of technology which has improved the feasibility of these systems will be described--the tools of computer system design and analysis. More specifically, the techniques of performance evaluation and modeling will be discussed in their relation to design of multiprocessor systems and research described by this thesis.

2.3.1 Techniques of System Design

The current single processor systems have only recently reached a complexity requiring sophisticated techniques of analysis, as witnessed by the growth of the area of performance evaluation in the last ten years. On the other hand, the design problems previously mentioned for multiprocessor systems make these systems so complex to begin with that careful analysis of these problems is mandatory. Therefore, most of the multiprocessor systems described were developed through the use of performance evaluation tools during the design stage. The empirical performance analysis techniques such as the instruction mix, the kernel, and the benchmark (2,56) are not of concern here; however modeling, both analytic and simulation, and monitoring will be discussed in detail.

While graph theory, mathematical programming, and decision theory have been used in performance evaluation, queueing theory has been the principal analytic tool (5,70,90). The state of a computer system is described by the specification of which jobs are in which of the operating system's queues. Correspondingly, the state of a network of queues is described by the specification of the number of customers in each queue. While the identification of individual tasks is lost in the queueing theory representation, the basic structure of this model is close to that of the real system. At first, it may appear that this relationship would provide the ultimate tool for performance evaluation. However, in practice, it has often been found that as more detail is added to these models, the assumptions necessary to provide a solvable model tend to reduce their validity.

The application of queueing theory to computer systems modeling

began in the early sixties. Surveys of this early work are available in (27,65). Basic to these models are assumptions about interarrival and service distributions. Several studies have been made to validate these assumptions (26,36,44). More recent work has begun to use specially developed queueing theory models in attempts to model the particular idiosyncracies of various batch and time-sharing configurations (21,62,87). While entire configurations can be modeled, this level of detail sometimes does not allow accurate investigation of particular components; so models are often made of a single component. This approach is taken, for example, in the study of paging (61,77) and memory interleaving (8,16). Several studies have been made to optimize particular aspects of computer performance. Buzen has developed a queueing network model of a multiprogramming system to analyze optimal assignment of requests to interchangeable resources of different 'speeds' (17). Sapiro has studied the possibility of controlling waiting time by optimizing the service rate (83). Several models of complex multiprocessor systems have been developed. A basic model uses a finite source queueing model to characterize the interaction between the problem programs and the operating system (66). Other work has built onto this basic model to provide a more realistic representation (29,81). The importance of the application of queueing theory can be seen in the fact that the first symposium by the Association for Computing Machinery Special Interest Group on Measurement and Evaluation consisted almost entirely of analytic models (37).

Simulation of complex computer systems by computer programs has evolved to be the most widely used technique of performance evaluation

(51). The flexibility of simulation allows researchers to investigate any matters of concern in any degree of detail. However, this power has its costs: The manpower required to develop simulation models is sometimes prohibitive to the degree that its cost approaches that of actual system development (9).

With no engineering experience in this area, designers of current multiprocessor systems have typically had to resort to simulation to examine the alternatives in structure (58,67). However, as multiprocessor systems increase in size, simulation will become difficult, and maybe even impractical (53).

One of the difficult problems in devising a simulation model is deciding which features of the actual system are relevant to the problem and should be included in the model. If a large amount of detail is included, the model will be more accurate, but it will also take longer to program, debug, and execute. On the other hand, if only major factors are considered then the program may execute relatively quickly; but its validity would be in doubt. The two major levels of detail are the microscopic--where the major unit or transaction is the instruction, and the macroscopic--where the major transaction is the task (97). Within each level, a minimal set of variables which characterize the system relative to the purpose must be developed. As input, some of these variables describe the workload and system configuration desired for a given analysis. As output, other variables describe the results of applying the workload to that configuration.

Numerous descriptions of attempts to simulate specific systems have pervaded the literature. Probably the most well known was the pre-

installation simulation of an IBM 360/67 at Stanford by Nielson (73,74). His simulator was used to determine how to alter the original system specifications to correct for a memory bottleneck. Since then, a variety of simulators have been written for specific batch and time-sharing systems (43,52,69,82), information retrieval systems (45), real-time systems (67,91), and specific operating system algorithms (88). One slightly more general simulator was designed to model any of a series of computers--the 360 series (57,58). Most of these efforts were reported as successful, but validation of their models was weak or nonexistent.

The structure of these models was usually event oriented. That is, a calendar of future events is maintained; and after one event has been simulated, time is advanced to the time of the next event. Here an event usually corresponds to an I/O interrupt (64). One well known simulator which does not fit this pattern is SCERT (48,55). Instead of simulating the events, a table of empirically derived data is examined to determine the probable results of the event. As with event oriented simulators, the accuracy of this method has been questioned (51).

Many of the simulators such as those mentioned above were written in general purpose simulation languages such as GPSS (95) in order to remove much of the burden of bookkeeping from the designer so he could put his efforts toward more constructive ends. However, it was found that these languages imposed restrictions which were cumbersome to circumvent (24). A simulation language based on FORTRAN, GASP, eliminated these restrictions by allowing the user to program in FORTRAN when necessary (80).

Several languages have been developed specifically for simulating computer systems (51). IBM has developed a language called Computer Systems Simulator, CSS, to evaluate System/360 hardware and software configurations (85). As with Lockheed's LOMUSS (52) and RAND's ECSS (74), CSS has special statements to simplify hardware and software specification. These languages, however, execute very slowly.

Simulation programs, whether written in FORTRAN or one of the special languages, have one invariable feature--garbage in, garbage out. While accurate representation of the system's configuration is not a perfected science, most "garbage" results stem from inadequate characterization of the workload. It is not enough to know that a certain scientific application is "compute bound"; specific information on how tasks behave is needed.

To this end, techniques of monitoring the activities of computers have been developed to trace the flow of tasks through the system and then provide this information to a simulation model. Trace-driven simulation modeling has proven to be the current most effective means to model complex computer systems for performance evaluation (23,75,92).

Besides providing input to simulators, monitoring techniques are effective in themselves for providing information for performance evaluation. Monitoring techniques are generally classified as either hardware or software. A hardware monitor is a device, external to the computer, which records the activity of the system by means of a number of probes attached to selected signal lines within the computer hardware. Types of things which can be determined in this manner include: processor idle time, overhead time, channel and device active time, and

processor-I/O overlap time. Excellent results have been obtained from research efforts using this type of monitor (1,3,4,11). A software monitor is a program which gathers data from operating system tables and registers. Unlike the previous method, software monitors degrade the performance of the system (64). However, they can provide detailed information which cannot be hardware monitored (10,50,59). Both methods require data reduction and reporting techniques (31).

For the most part, the decision to monitor has been made after the system was completely designed and running. Much effort was therefore put into designing monitoring techniques which did not require much alteration in the system. The techniques developed through these efforts made it feasible to incorporate monitors into the basic design of new systems (13,19,20,71). With these new systems, it is possible to trace various events to various levels of detail. With real-time output, operators have been able to spot trouble areas and initiate corrective measures (86).

2.3.2 Performance Design

Some of the techniques of performance evaluation have been discussed. Initially, it was stated that these tools were necessary for analysis of alternative structures during the design of multiprocessor systems. However, there is an even more important place for these techniques in the design of computer systems--an area which we refer to as performance design. Performance design is concerned with the use of performance evaluation techniques as an integral part of the structure of the operating system in such a way that the operating system uses the techniques to monitor the computer system's performance and dynam-

ically alter the system to make it more responsive to the immediate requirements of the workload.

The real-time output of monitored information has provided an important first step in this area, but the analysis of this information must be performed by the computer, not the operator. There appear to be two approaches to providing the operating system with the capability to perform this analysis. The first approach consists of providing the operating system with a set of heuristic rules based on empirical knowledge gained through the investigation of computer system principles by simulation. As queueing theory develops, a second approach will become available: providing the operating system with the capability to predict theoretically the proper action of certain portions of the system. Thus, the further development of the techniques of monitoring, simulation and analytic modeling are crucial to the future development of a dynamic operating system.

Performance design is the approach taken in this research to solve a specific problem in multiprocessor design which is described in the next section.

2.4 Studies on the Problem of Supervisor Queueing

Recall that the two design alternatives for the control of multiprocessor systems are master-slave and floating executive control. In both of these cases, a job generates an interrupt which causes a request for the supervisor. If the supervisor or its data base is in use at the time such a request is made, then that request is queued until the supervisor becomes available. An important point is that while such a request

is waiting on the queue, the processor which was executing the interrupted job remains idle. The more a processor waits on the supervisor queue, the more this idleness decreases the system's throughput. The effect this queueing has on the throughput for a two-processor system is probably small; but as the number of processors increase, the effect could conceivably become substantial. Studies which analyzed the magnitude of this effect are described below.

2.4.1 Simulation Modeling

A simulation study of a multiprocessor system has been made by Lehman (6). His main concern was with memory interference, but he did analyze supervisory queueing to some extent. However, the results of his simulation are not directly applicable to true multiprocessor systems because his model was of a large matrix multiplication problem executed in parallel--one row's operation on each processor. His results indicate that processors were idle 0.8% of the time because of supervisor queueing in a system which had 16 processors and 64 memory modules during the matrix multiplication.

2.4.2 Queueing Theory Modeling

Madnick (66) interpreted the standard finite source queueing model with quasirandom input to express the performance degradation in terms of three parameters: E, the average time a task processes between interrupts; L, the average time the supervisor is held by a processor to handle an interrupt; and N, the number of processors.

The flow of jobs through his model could be represented as in Figure 1. Jobs are selected from an infinite population, a, which have exponentially distributed processing times and are put into the proces-

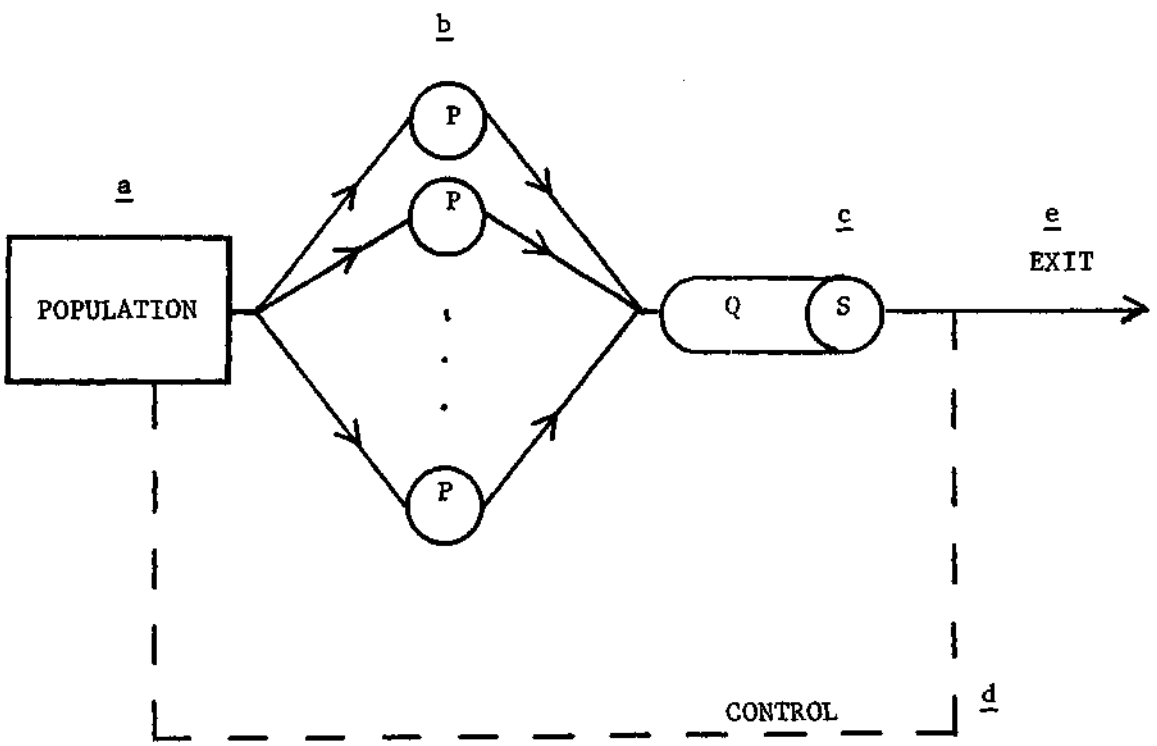


Figure 1. Job Flow for Madnick's Model

sors, b. At the end of the processing time, the job requests the use of the supervisor, c, to handle an interrupt. At the end of the supervisor's exponentially distributed service time, the job triggers the random selection (from a) of another job for the processor, d, and then leaves the system, e.

While this model ignores many features of real systems, it does seem to represent the essential characteristics of the supervisor queueing problem.

The model is based on a set of steady-state probabilities P_i, i=0,...,N. Each probability P_i corresponds to a state of the system S_i, which represents the case when i processors are attempting to use the supervisor (one processor using it and i-1 on the queue). A queue exists when the queueing system is in any of the states S_i, i=2,...,N. If the system is in state S_2, then the queue length is one; if the system is in state S_3, then the queue length is 2; etc. Thus, the average queue length is

$$Q = \sum_{i=2}^{N} (i-1)P_i \quad , \tag{1}$$

where

$$P_i = \frac{N!}{(N-i)!}\left(\frac{L}{E}\right)^i P_0 \tag{2}$$

and

$$P_0 = \left[\sum_{i=0}^{N} \frac{N!}{(N-i)!}\left(\frac{L}{E}\right)^i\right]^{-1} \tag{3}$$

Equations (1), (2), and (3) provide an expression for Q in terms of N, L, and E:

$$Q = \frac{\sum_{i=2}^{N} \frac{(i-1)}{(E/L)^i (N-i)!}}{\sum_{i=0}^{N} \frac{1}{(E/L)^i (N-i)!}} \qquad (4)$$

The derivation of (4) is available in any queueing theory text (30), with E and L usually expressed as $1/\gamma$ and $1/\mu$ respectively. Based on monitored data, Madnick concluded that a reasonable estimate of L/E would be between .001 and .010.

This model indicates that the performance degradation due to supervisor queueing is significant. For example, based on L/E = .05 and 21 processors, an average of 2.8 processors would be idle. Complete elimination of this degradation, if possible, would increase throughput by 16%. Other authors besides Madnick have also discussed the severity of this problem (40,67).

This research will seek to reduce this degradation by scheduling, as described in this next chapter, using the concept of performance design.

2.5 Survey of Current Scheduling Techniques

2.5.1 Workload Independent Techniques

The previous survey of current multiprocessor systems demonstrated that schedulers for multiprocessor systems are currently much like those for single processor systems. They are based on a first-come-first-

serve (FCFS) or roundrobin search of the mix with some form of priority and usually a time-slice (96). The requirements for a job to enter the mix are usually simply core and device requirements as specified in the job control language. Except for these requirements, the selection of jobs to process is, in general, independent of the characteristics of the jobs.

2.5.2 Workload Dependent Techniques

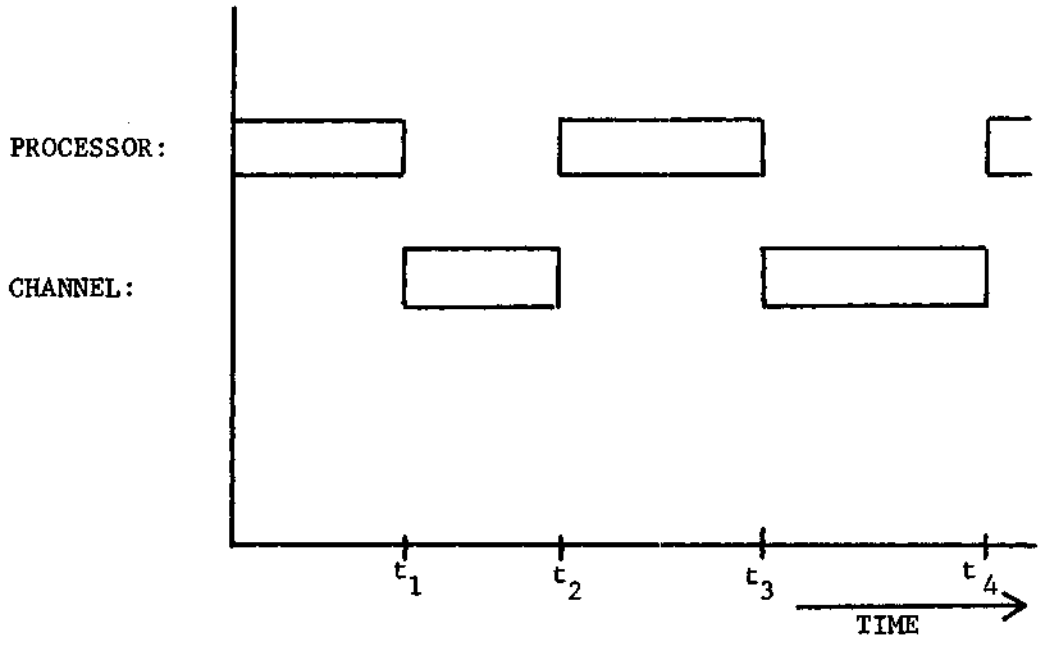
Recently, some schedulers have begun to base their selection on other job characteristics. One of these has been tested for use on one of the multiprocessor systems previously described (76). The job characteristics considered here are: 1) CPU time, 2) number of tape drives, 3) number of input cards, 4) programming language 5) number of disk files, and 6) number of output pages. The scheduler uses this information to assign jobs to classes which indicate their ability to utilize the system. When the scheduler is activated to select a job, it will first select a class which has characteristics which will make the total workload meet some performance criteria, like load balancing. Then a job is selected from this class. While the characteristics of each class are initially determined from information on job control cards, they are updated based on the class's actual running characteristics. Thus this scheduler is a straightforward, but sophisticated, extension of selection for mix entry based on device and core requirement.

IBM's VS2 Release 2 uses a scheduling algorithm aimed at workload balancing (54). Jobs are placed into performance groups according to the demands which they are expected to place on the system. The scheduler then selects jobs according to two basic requirements. Jobs are

selected from performance groups which are expected to balance the workload and such that the various performance groups meet predefined levels of activity. This scheme, however, is not as sophisticated as the one previously described because jobs are manually placed into performance groups through a special control card parameter and the group's attributes cannot be modified when the jobs' actual demands on the system do not meet the demands expected for that performance group.

Another scheduler tested for the Burroughs B5700 dual processor system was based on quite a different characterization of the workload (79). For each job, when the job was swapped out, processor utilization by that job was computed for the period during which the job was swapped in. This was done each time a job was swapped in and out, providing a series of samples of processor utilization. This series was used to forecast, using a dynamic double exponential smoothing formula, the expected processor utilization which would occur the next time the job was scheduled to run. The description here has been for processor utilization, but similar statistics were also gathered on core and I/O utilization. Based on these three job characteristics the scheduler would select a job which would provide the optional system utilization.

Another way to characterize jobs in the workload would be thru a flow chart representation, as in Figure 2. This representation is based on the fact that a job's activity can be broken down into two categories: processing and I/O. A job will process until an I/O instruction causes an I/O initiate interrupt to be generated. It will then perform I/O until completion, at which time an I/O complete interrupt will be generated. This procedure will be repeated with alternat-



t_1, t_3, . . . : I/O INITIATE INTERRUPT

t_2, t_4, . . . : I/O COMPLETE INTERRUPT

Figure 2. Flow Chart for One Job

ing processing and I/O cycles.

Several factors have been ignored in this description. First the activity of the supervisor, and of other jobs, has not been represented. This was left out because Figure 2 represents only one job, not the entire workload. Another point which needs to be clarified is that not all I/O instructions initiate I/O device activity. Actually an I/O instruction will initiate I/O activity only if previous I/O instructions have filled the I/O buffer. Therefore, the I/O cycle would perhaps be better interpreted as the time during which a job is blocked from processing due to I/O activity. Likewise, the processing cycle could be interpreted as the length of time a job processes before being blocked from processing due to an I/O activity.

This representation is extremely useful because it captures the points of allocation and de-allocation of two major system resources: processors and channels.

Representations similar to this have been used in several studies (43,63,82,89). One of these represents a job only by its processor cycles (89). Through an intercept software monitor in the interrupt handler, the length of each processing cycle of all jobs is monitored. Based on a job's history of processing cycles, an estimate of the length of the next processing cycle would be determined through a forecasting technique for every job in the mix. The scheduling algorithm would select a job which had the shortest predicted processing cycle.

The processor-I/O cycle representation will be used to support a scheduling algorithm, described in the next chapter, which aims at minimizing the problem of queueing of requests to the supervisor.

CHAPTER III

REDUCTION OF SUPERVISOR QUEUEING THROUGH PERFORMANCE DESIGN

3.1 Clustered Resource Scheduling

A specific problem in multiprocessor design, queueing of request to the supervisor, has been identified and explained. The concept of improving system performance through the incorporation of some tools of performance evaluation into the operating system has been introduced. The technique of using the scheduling algorithm to implement performance design into the system has been supported by citing examples.

The rest of this thesis will deal with the design and analysis of a scheduling algorithm based on performance design which will attempt to reduce performance degradation caused by queueing of requests to the supervisor in a large multiprocessing system.

3.1.1 The Basic Scheduling Algorithm

A natural solution to the problem of processor lockout would be to schedule tasks to processors such that a processor would request the supervisor at a time when no other processor needs it (67). Since a processor requests the supervisor when its task generates an interrupt, implicit in this solution is the assumption that, for every task in the mix, the length of time until each task generates its next interrupt is known. While this is not generally known, recall that this information could be forecast from previous processing cycles of the job, as in (89).

Another assumption implicit in the above solution is that the length of time a processor holds the supervisor in order to handle an interrupt is known. This, too, is not known, but a reasonable estimate could probably be made for each type of interrupt. Such an estimate could be based on either recent history of that type of interrupt, or on supervisor instruction timings and knowledge of the lengths of the various queues which the supervisor must search for any particular type of interrupt. This research will assume that this information is available.

The algorithm to implement this solution could be expressed as a two-table search (47). Table 1 would have an entry for each ready job in the mix specifying the time until the next interrupt and the supervisor time required to handle that type of interrupt. Table 2 would have a schedule of supervisor idle periods which would indicate when, in the near future, the supervisor has been predicted to be available. When the supervisor finished handling an interrupt, then a task would be scheduled for the processor released by the interrupted task. To find a task, Table 1 would be searched in an order specified by task priority, or some other external criteria. For each task, a decision would be made as to whether the supervisor had an idle period corresponding to the period from the current time plus processor cycle time to the current time plus processor cycle time plus supervisor time. A match would cause the task to be scheduled for the processor and the period when the task would cause the processor to use the supervisor to be eliminated from the table of supervisor idle periods. This algorithm will be hereafter referred to as Clustered Resource Scheduling (CRS).

3.1.2 Example Situation

For example, consider the period of time represented in Figures 3 through 7. Figure 3 shows the supervisor schedule. The S's indicate periods of supervisor time that have already been allocated. The current scheduling point is at the end of one of these periods. Figure 4 represents the selection of a job which will process from the current schedule point for some length of time, P_1, and then use the supervisor for some length of time S_1. The next schedule point will, of course, correspond to the end of the next supervisor period. At this point, a job will be scheduled which will process for P_2 time units and hold the supervisor for S_2 time units, as in Figure 5. Figure 6 shows that a job of processing length P_3 could not be scheduled at the current schedule point because its request for the supervisor would overlap with a previously scheduled request. An alternative selection, P_3', is illustrated in Figure 7.

3.1.3 The Modified Scheduling Algorithm

If this algorithm could always find a job to meet the processing cycle length and supervisor service length requirement, and if the predicted information was accurate, then unnecessary queueing of requests to the supervisor could be eliminated. Recall that according to Madnick's queueing model, this could mean a 16% increase in throughput in a large multiprocessor system, corresponding to a very large additional amount of work.

However, if the scheduler must always find a job in the mix which meets both requirements, then the mix may have to be very large. If the mix is very large, then a search of the mix could be very time-consuming.

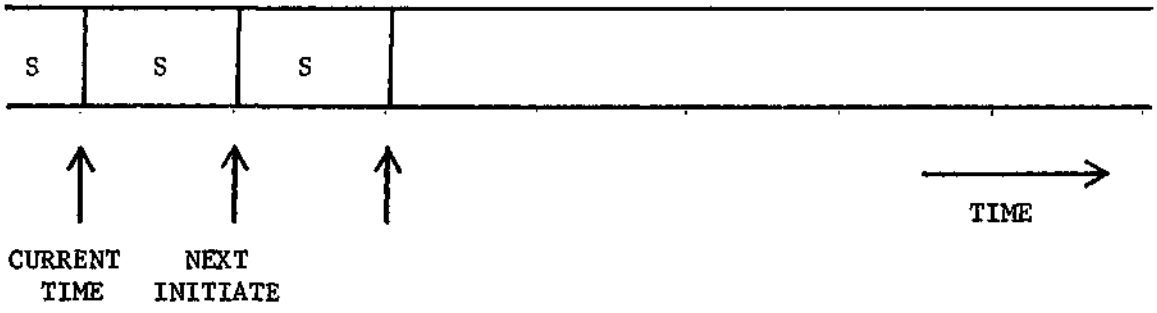


Figure 3. Supervisor Schedule

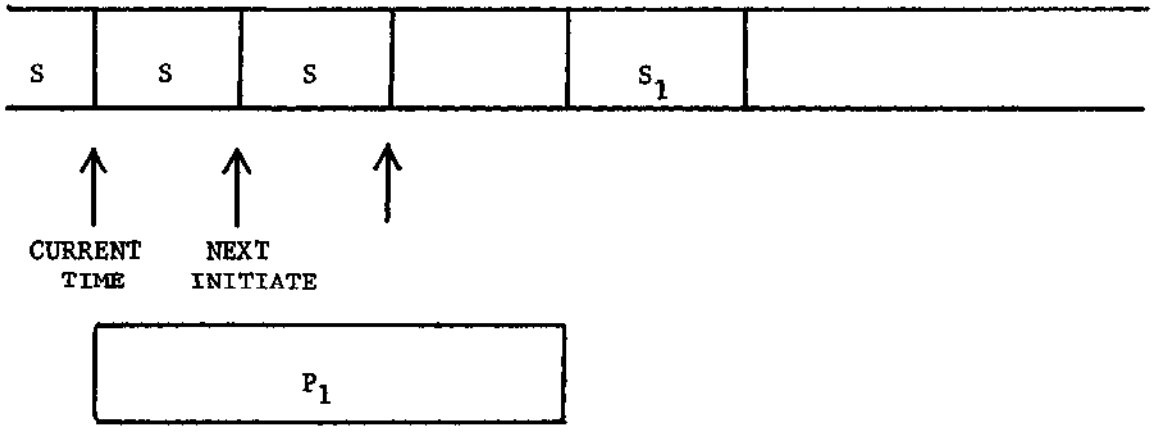


Figure 4. Scheduling Job P_1

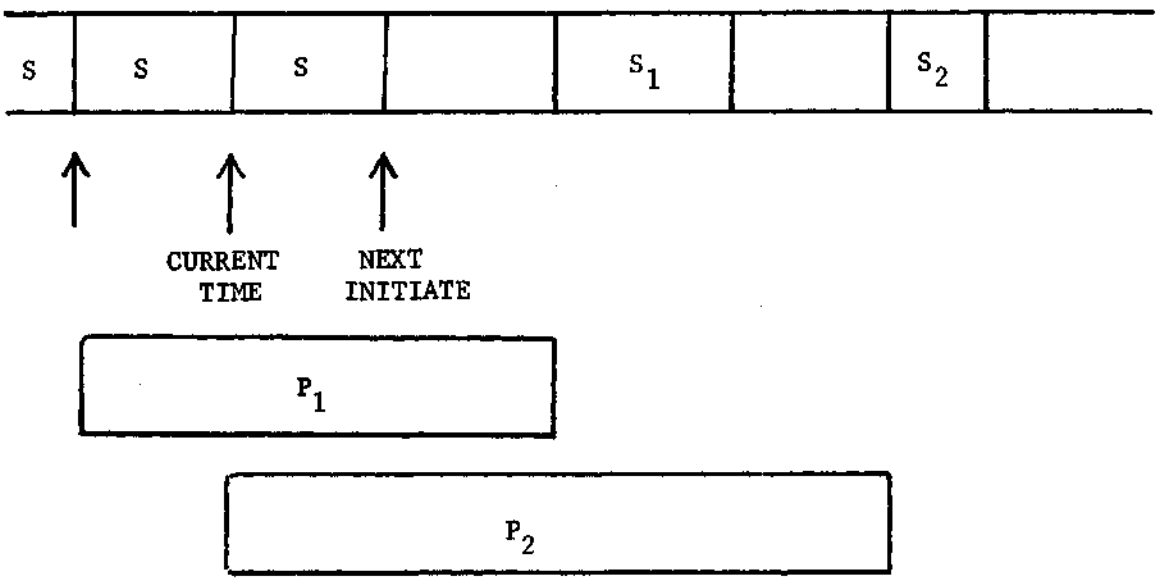


Figure 5. Scheduling Job P_2

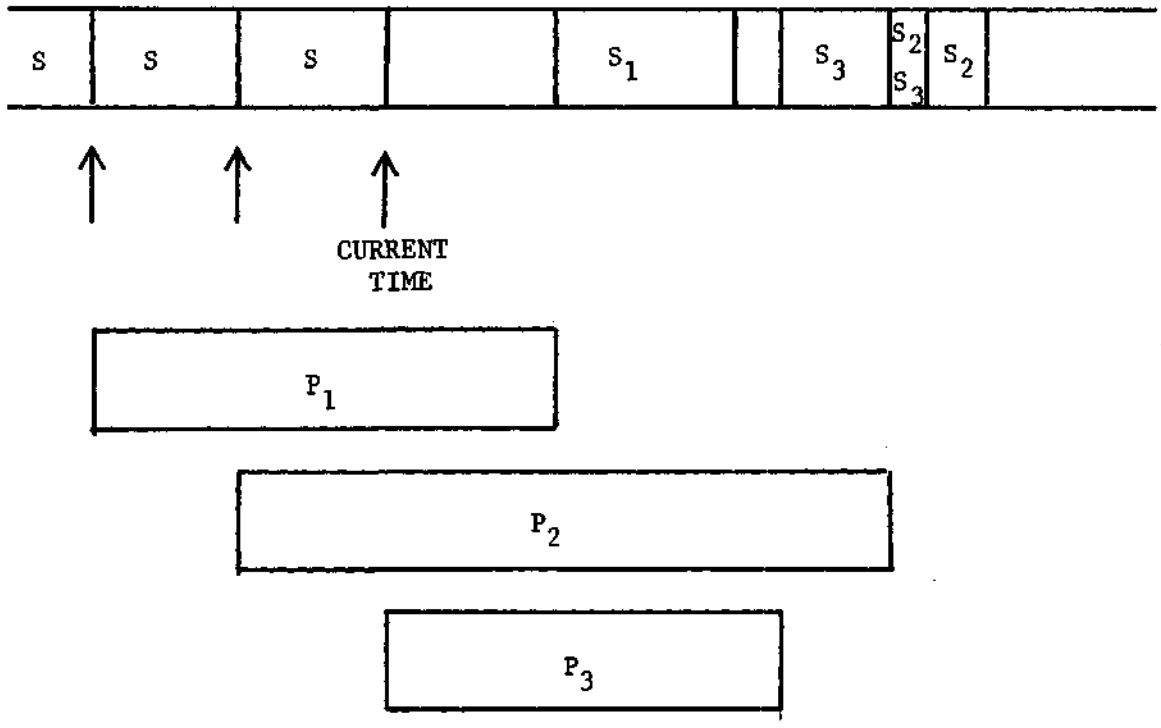


Figure 6. Attempt to Schedule Job P_3

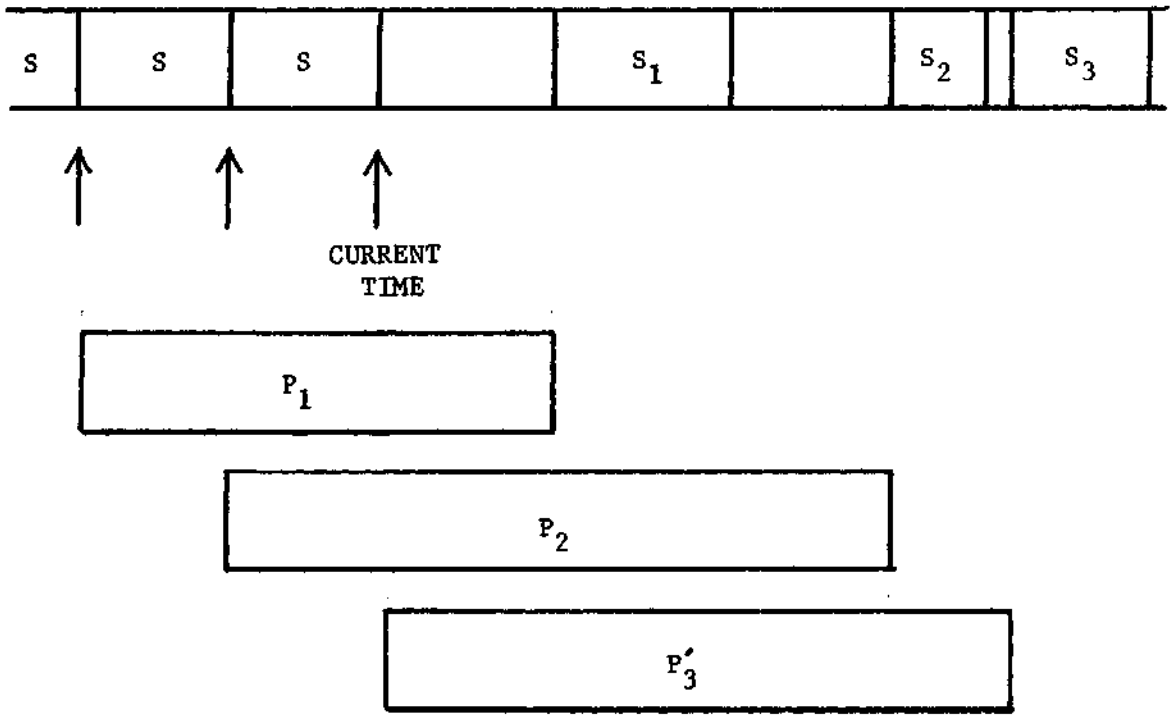


Figure 7. Proper Scheduling of Job P_3'

Thus it becomes questionable whether the algorithm as described above would be cost-effective. If this algorithm would take twice as long to execute as, perhaps, FCFS, then the load on the supervisor would be doubled; and this would probably overshadow any benefits gained through the algorithm.

With this problem in mind, a modified algorithm has been designed that is based on a parameter which provides a tradeoff between the percentage of maximum improvement achievable and the efficiency of operation.

The modified algorithm is similar to the original except that supervisor time is allocated in integer multiples of a fixed block of time. The block size is the parameter which provides the tradeoff. A lower limit on the block size is one time unit. This case would function exactly like the original algorithm. As the block size increases the algorithm would become faster, but the effectiveness in reducing queueing would be reduced. The cause of the reduction in effectiveness is illustrated in the following example.

Consider the case where the block size is equal to the average supervisor service time. Consider, also, the additional restriction that one and only one block of supervisor time will be allocated to each supervisor request. With these constraints, the same data used in the previous example is used in Figures 8 to 11 to illustrate the modified algorithm.

3.1.4 Example Situation

Figure 8 represents the supervisor's schedule marked off by blocks. In Figure 9, the selection of a job which will process for P_1

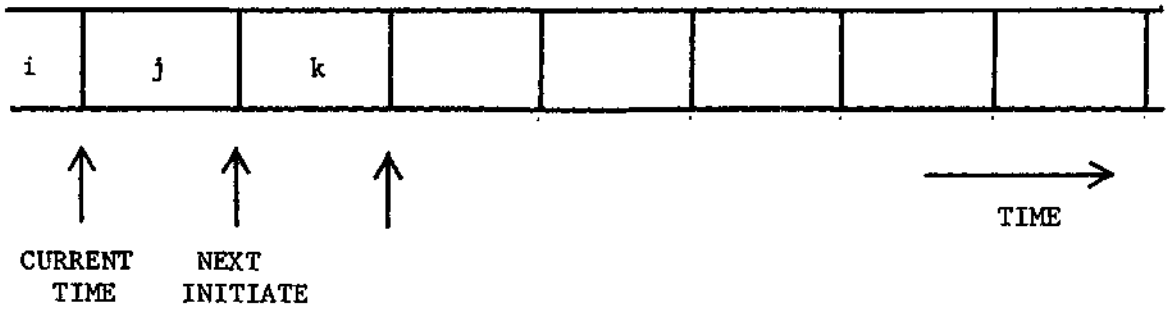


Figure 8. Blocked Supervisor Schedule

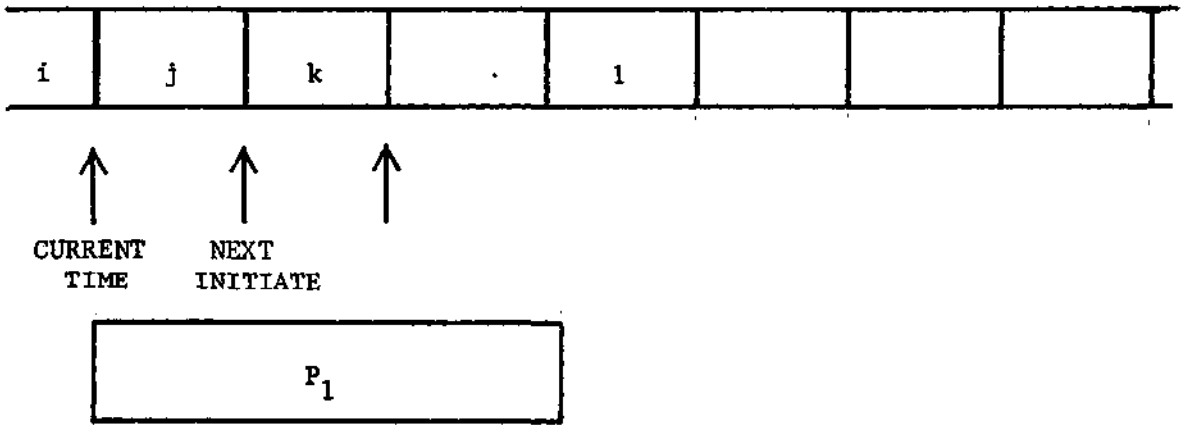


Figure 9. Blocked Scheduling of Job P_1

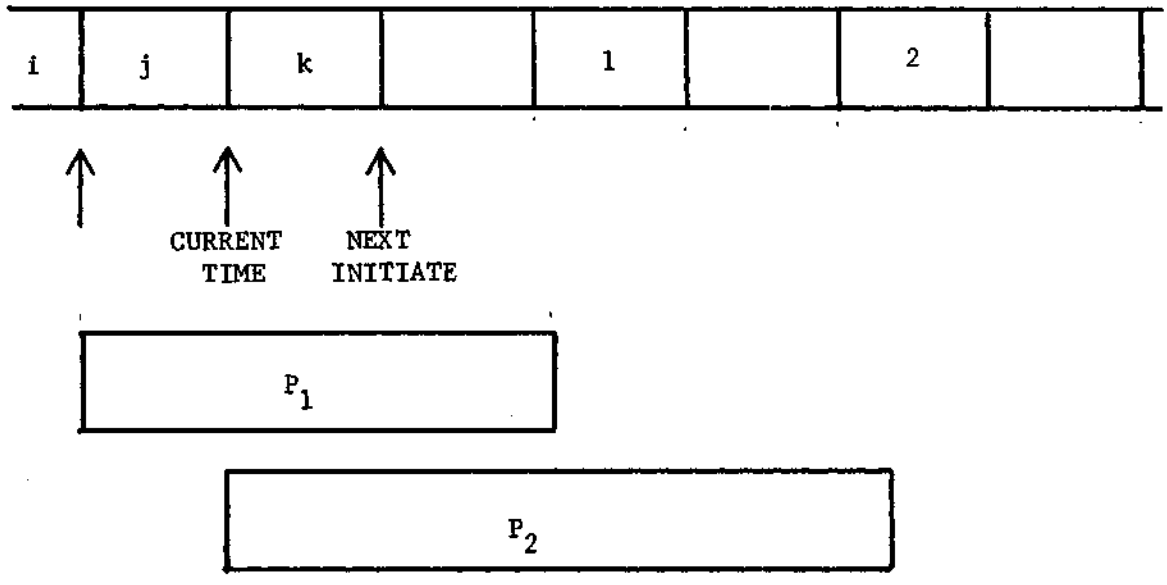


Figure 10. Blocked Scheduling of Job P_2

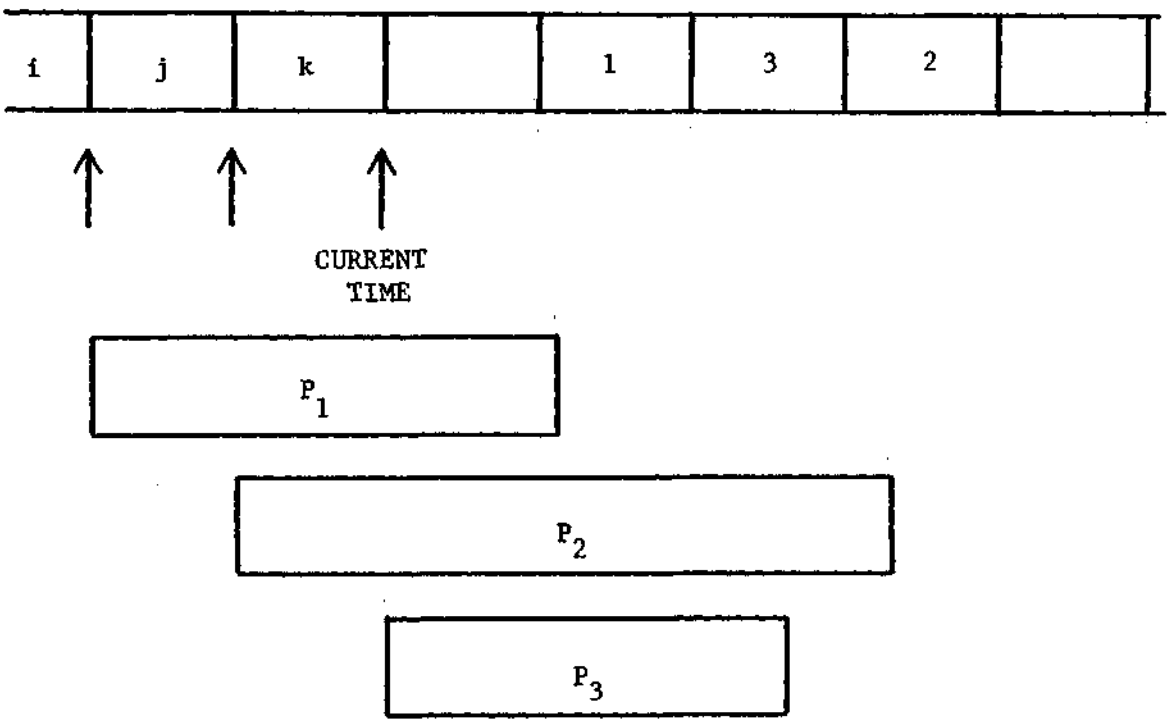


Figure 11. Inaccurate Scheduling of Job P_3 Caused by Blocking

time units is accompanied by an indication, 1, that the block of supervisor time during which the processing period ends is no longer available. A processing period of length P_2 is represented in Figure 10, just as was done in Figure 5. Figure 11 shows that a processing period length of P_3 could be selected next. Recall that in Figure 6, the original algorithm would not allow this because the supervisor utilization would overlap. The resulting supervisor queueing degrades the effectiveness of the algorithm.

This example has introduced two basic algorithm parameters: block size and maximum number of blocks allowed to be allocated for any one supervisor request. These and other parameters necessary for the complete specification of the algorithm will not be detailed here, but rather will be discussed in Chapter 4 through a series of experiments which guided the development of the complete algorithm.

It should be pointed out that the supervisor schedule could be implemented simply as a one-dimensional array, with each element in the array corresponding to one block of time. The array would have to be large enough so that the largest processing cycle could be scheduled. It would not have to be infinite since the block corresponding to the last element of the array could be followed by the block corresponding to the first element of the array, as in a circular list.

3.2 Simulation Model Description

3.2.1 Simulation Approach

Two simulation models were developed for the study of this algorithm. A preliminary study used a model programmed in GPSS which was

based on Madnick's model of a computer system. This model allowed two scheduling algorithms: First-Come-First-Serve, and CRS. The purpose of this model was to determine if there were any special problems or considerations which should be taken into account in the design of the second, more detailed model.

The second model, which was programmed in GASP, was implemented for this study on the Burroughs B5700 operated by the School of Information and Computer Science at Georgia Tech. This simulator was based on a more realistic computer system model.

3.2.2 Computer System Model Flow Diagram

Just as Madnick's model was described by the Job Flow Diagram in Figure 1, the GASP model is described in Figure 12.

The parameters of a job's processing cycle distribution are selected from a specified population, and various other job characteristics are selected from their distributions, a. A specified number of these jobs are placed in a finite mix, b. A job is selected from the mix in a specified order by a specified scheduling algorithm, c, and is placed in an available processor, d. Meanwhile, the mix position it left remains vacant until the job returns to the mix from point m. At the end of the job's processing cycle, it requests the use of the supervisor, e. After the supervisor's service time, the scheduler is triggered to enter a job from the mix into the available processor, f. Also, the job which was using the supervisor does one of two possible things. If it has completed all of its cycles, it exits the system, g; then another job is triggered to enter the mix position made available, h. If it has not completed all of its cycles, it proceeds to the channel queue, i, to

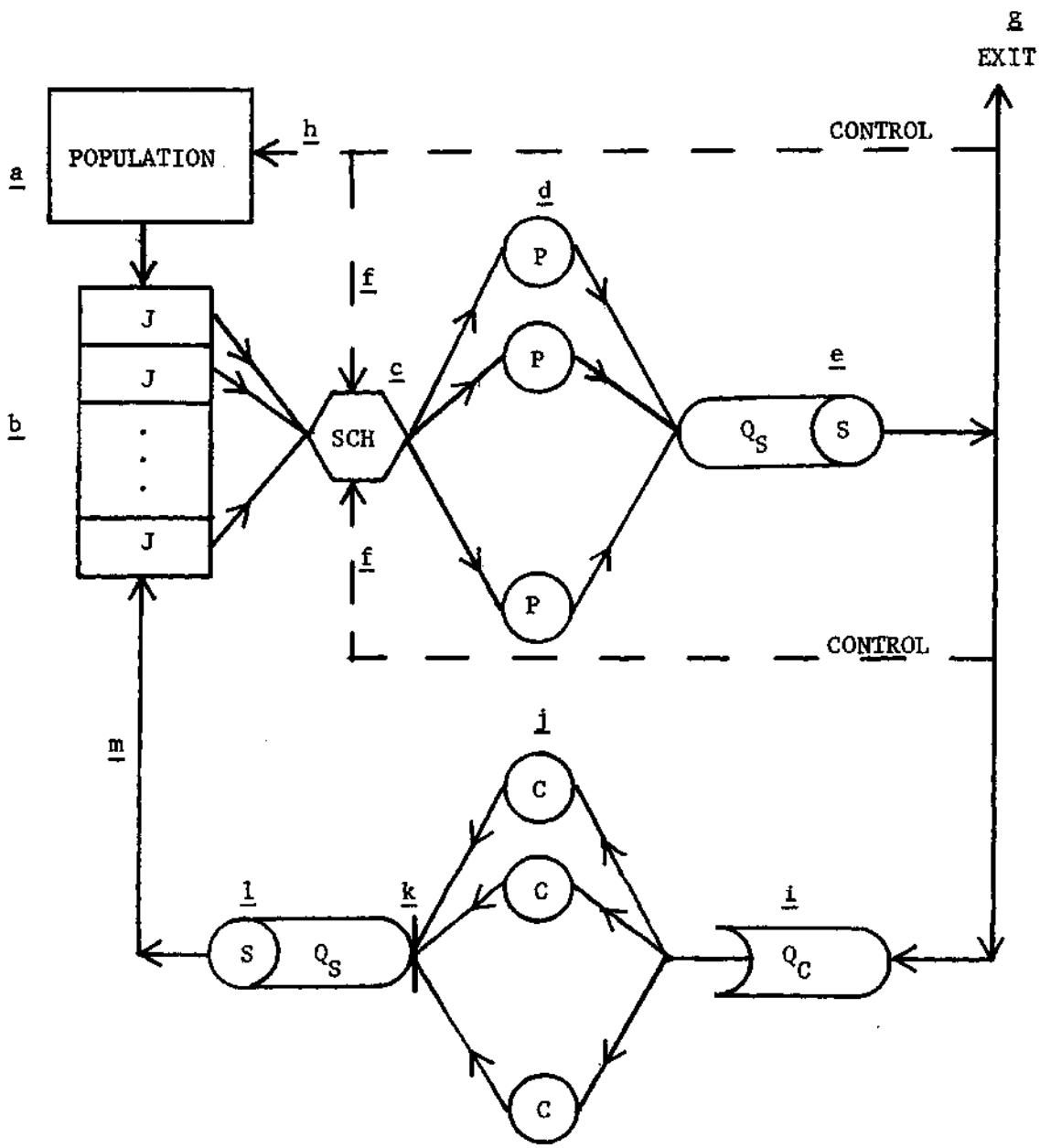


Figure 12. Job Flow for GASP Model

wait until a channel, j, becomes available so that the I/O cycle may be processed. After completion of a job's I/O cycle, the job goes to point k where several alternatives are available which deal with the method of handling the I/O complete interrupt. These will be detailed later. In either case, when the job leaves the supervisor, l, it re-enters its mix position, m.

While this model is more realistic than Madnick's model, it is not meant to model all features of computer systems. However, the model should be accurate enough to give a reasonable indication of the effectiveness which CRS would have in a real system. The major difference between Madnick's model and the detailed model is with respect to the input process to the supervisor. The input process for the detailed model is more restricted than that for Madnick's model. This restriction is, of course, caused by the finite mix and the I/O cycle delay. The effect of this restriction should be a decrease in the variety of jobs available for selection. Accurate representation of this factor is essential for any study of a methodology which is dependent on such variety.

3.2.3 Model Structural Alternatives

The system control design alternative assumed in this model is floating executive control. Furthermore, it is assumed that all the parts of the supervisor represented in this model utilize the same LOCK-UNLOCK flag. The prior assumption is desirable and the latter, reasonable. For example, consider the following straightforward attempt to break the supervisor components modeled in this study into distinct sections: 1) routines which handle I/O initiate bookkeeping, 2) rou-

tines which handle I/O complete, and 3) routine which schedules jobs. This breakdown would be infeasible since all three categories usually access the mix.

Other attempts to divide the data base into distinct parts would meet similar problems. There is necessarily a large amount of interaction among the various functions of the supervisor in order to provide each with knowledge of what the other is doing. Thus a single LOCK-UNLOCK flag associated with the data base not only eliminates many operating system design problems but also simplifies the basic structure of the supervisor and correspondingly reduces the problem of system debugging.

The memory design is assumed to be an interleaved multi-module core-resident design with crossbar connection with processors, such that physical memory interaction does not affect performance. This assumption could be replaced with the assumptions that the supervisor resides alone in one particular module, and that memory interference has a uniform effect on problem program performance. With either assumption, the problem of memory interference can be ignored.

A third design alternative involves the technique for assigning processors to handle I/O complete interrupts. Prior to considering several alternatives, it should be pointed out that no mention has been made of allocating part of the supervisor's time to handle I/O completes. Clustered Resource Scheduling was not applied for this case for two reasons. First, it may be considerably more difficult to forecast the I/O cycle length than the processor cycle length. A large component of the I/O cycle is usually rotational delay. The variance in this com-

ponent is probably at least an order of magnitude larger than the total time required to handle an I/O complete. A second reason for not scheduling I/O complete usage is that an I/O complete service time is much smaller than an I/O initiate service interval, and therefore may not drastically degrade performance. For example, consider the situation where the block size equals the average supervisor time and the maximum number of blocks allocated for any I/O initiate is one. If you assume that every block is eventually allocated, then an I/O operation will be initiated at every block. From this it is reasonable, but not completely accurate, to assume that, on the average, one I/O complete will occur during each block. If this was the case, then the proper block size would be the sum of the I/O initiate service time and the I/O complete service time. Thus, the I/O complete actually does not need to be accounted for directly in the supervisor scheduling.

The problem of selecting a processor to handle an I/O complete interrupt would be eliminated if channels were designed so that they could handle these interrupts. It would not be unreasonable to require channels to reassign themselves to the next request at the completion of the current I/O operation. After all, processors do their corresponding reassignment.

Assuming the above capability is not available, this model handles I/O completes in two basic ways. The first one uses an algorithm which determines a processor to interrupt to handle the I/O complete: select at random, select the processor with the longest time remaining until its I/O initiate interrupt, or select the processor with the shortest time remaining until its I/O initiate interrupt. The second technique

is to delay the handling of the I/O complete until the next I/O initiate occurs. This last alternative is desirable because it reduces the amount of processor switching between supervisor state and problem program state and correspondingly reduces the number of times the scheduling algorithm must be called. However, this assumes a nominal average delay; any excessive delays would have to be controlled by some method.

3.2.4 Simulation Parameters

Besides the I/O complete algorithm, other model input parameters include: mix size, number of processors, number of channels, and scheduling algorithm. The three scheduling algorithms available are CRS, FCFS, and Round Robin. As the concept of a time slice may not be valid for large multiprocessor systems, this is not explicity included in any scheduling algorithm. Also, the distributions for processing cycles, I/O cycles, numbers of cycles, priority, I/O initiate service time, and I/O complete service time must be specified. Each of these distributions is specified for the mix as a whole except for the processor cycle distribution. For that one, each job in the mix can follow its own, different distribution.

This last capability is important, as it is related to the major difference between the GASP model and Madnick's model that was mentioned in section 3.2.2. To represent fully the restrictions introduced by a finite mix, it is important to consider the effect of characteristics of individual jobs in the mix. If a particular job has the characteristic that it always has short processing cycles, then the mix position associated with that job will provide the scheduler with only short cycles for possibly a very long time, thereby affecting the variety of

the jobs in the mix.

A large number of statistics are gathered during simulation, many of which were motivated by the preliminary study. Each time a job finishes processing, the following information about that job is printed:

1) job number;
2) number of processing cycles, assuming every job starts and ends with a processing cycle;
3) priority;
4) total processing time--sum of processing cycle lengths;
5) total I/O time--sum of I/O cycle lengths;
6) arrival time--time at which job first entered the mix;
7) exit time--time at which job left mix, i.e., current time;
8) total ready time--total time during which job was available to process, but was not processing.

Also, for the purpose of determining when statistical equilibrium is reached, the average supervisor queue length is printed at every job exit.

At the end of the simulation the following information is printed:

1) average (minimum, maximum, and standard deviation of) length of supervisor queue and a histogram of the number in the queue versus the percent of time which that number were in the queue;
2) average (minimum, maximum, and standard deviation of) length of channel queue;
3) average number of processors busy;

4) average number of channels busy;

5) percent of time that the supervisor was busy;

6) average (etc.) processor cycle length generated by random number generator;

7) average (etc.) I/O cycle generated;

8) average I/O initiate service time generated;

9) average (etc.) waiting time of a job on the supervisor queue;

10) average (etc.) waiting time on the channel queue;

11) average processor cycle length of jobs selected by scheduling algorithm;

12) histogram of processor cycle selected (in terms of the number of blocks) versus the number of these selected;

13) histogram of processor cycle selected (in terms of the number of blocks) versus the average time that they waited in the ready state before being selected;

14) histogram of processor cycle lengths of ready jobs in the mix (in terms of number of blocks) versus the number of these (This could also be printed at specified intervals during the simulation.);

15) average number of different processor cycle lengths (in terms of the number of blocks) of ready jobs in the mix;

16) total problem program processing done;

17) total I/O done;

18) throughput: (16) + (17).

3.2.5 Special Simulation Considerations

In an experimental study, it is important to evaluate results only after proper use of statistical tests of confidence. Just as important is the proper design of a simulation model. A concerted effort was made to insure a statistically valid design.

First, a random number generator was chosen which had been thoroughly tested (63). In the previous section, it was mentioned that every time a job exits the mix, the average supervisor queue length was printed. This can be used to insure that statistical equilibrium is reached. There is a set of values for input parameters which reduces the full model to Madnick's model. This provides a means for validating the simulation model.

Two variance reduction techniques were also incorporated. The model was designed to insure that the same sequence of job parameter values would be generated when different design alternatives were compared--a separate random number generator was used for each different job parameter. For example consider the alternatives of using a normal versus a uniform distribution for the I/O cycle. The technique for generating a normal sample requires six random numbers while a uniform sample requires only one. If one random number generator was used for all parameters, then changing the I/O cycle distribution would, for example, also affect the number of cycles of jobs. The second variance reduction technique is the option of using the antithetic random number sequence.

3.3 Preliminary Considerations

3.3.1 Results of Queueing Model

The graphs in Figure 13 represent the expected supervisor queue length as a function of the number of processors in the configuration for several values of L/E, based on Madnick's model (66). One important point is made clear by these graphs: For each value of L/E, there is some number of processors such that adding one more processor causes an increase in the average supervisor queue of one. That is, after this point, adding an additional processor will provide no increase in throughput.

The cause of this effect is that the supervisor becomes 100% busy at this point. System designers must be careful to take this factor into account. It would be very poor design if a 16 processor system was controlled by a supervisor which could not support but 12 processors before becoming a system bottleneck. No type of scheduling could eliminate the idleness caused by such design.

3.3.2 Supervisor Utilization

Chapter II expressed the average queue length, Q, in terms of E, L, and N. An alternative form of this is (49):

$$Q = N - \frac{L+E}{L} \quad (1 - P_0) \tag{5}$$

Since P_0 is the probability that the supervisor is idle and has no queue, $(1 - P_0)$ is the probability that the supervisor is busy, i.e., the supervisor utilization.

$$(1 - P_0) = \frac{L\ (N-Q)}{L+E} \tag{6}$$

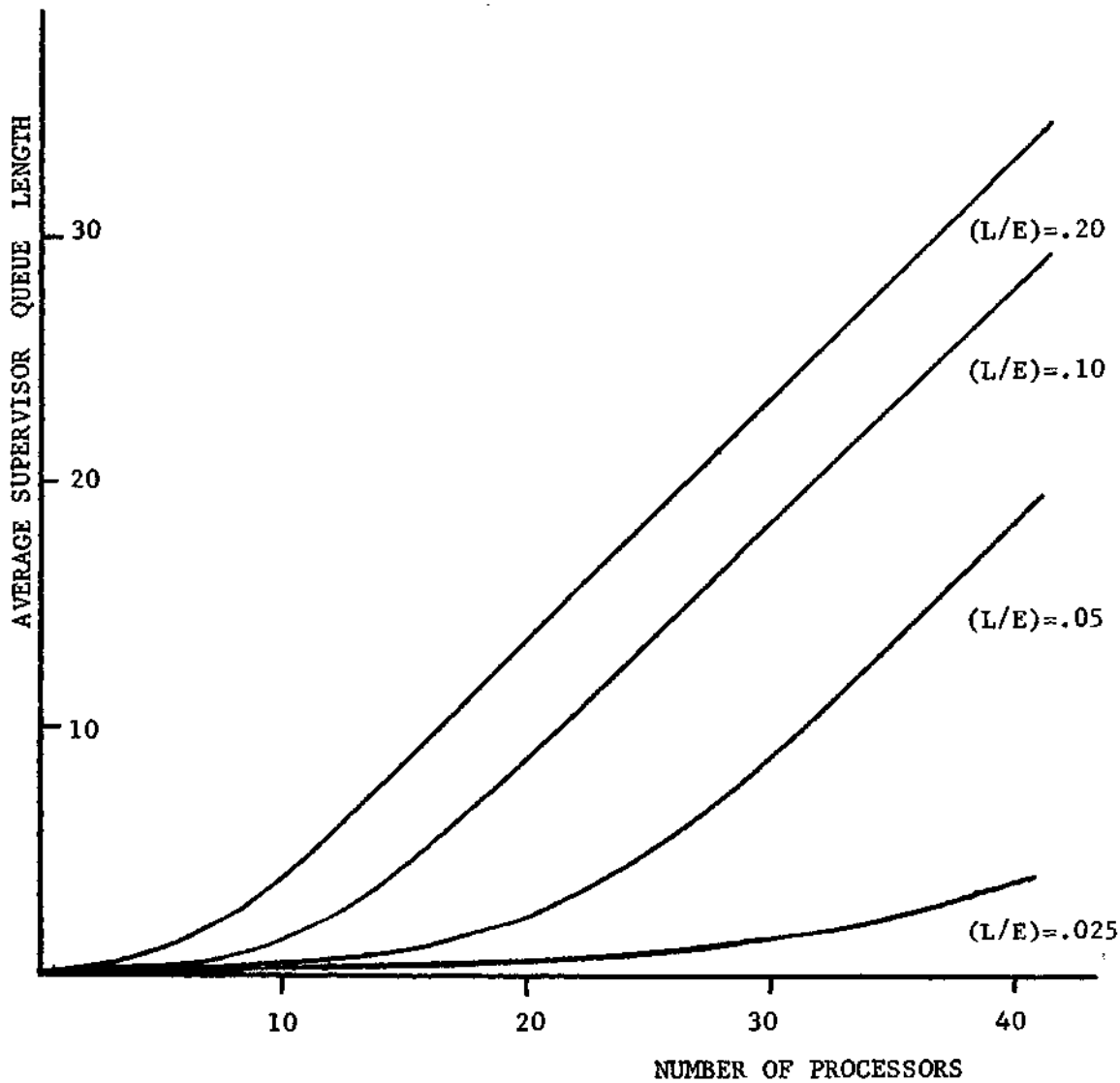


Figure 13. Queueing Theory Results

A previous example stated that if L/E = .05 and N=21, then Q=2.8. For this case,

$$(1-P_0) = \frac{(21-2.8)}{21} = \frac{18.2}{21} = .87 = 87\% \qquad (7)$$

3.3.3 Supervision Queueing and Randomness

Thus, in the above case, there exists a queue but the supervisor is not totally utilized. To understand this situation, consider Figures 14 and 15. The arrows indicate the arrival of a request for the supervisor, and the blocks represent the amount of time used by the supervisor to handle the requests.

In Figure 14, the first request occurs at time S and that request is serviced from time S to just prior to time U. At time U a second request occurs which is serviced from U to just prior to W. Again at time W a third request occurs which is serviced from W to just prior to Y. Assume that the sequence of events just described repeats itself, starting at time Y. If the time between any two adjacent letters (i.e. S-U, U-W, etc) is considered a time unit, then this example could be described by the parameters: N=3, L=2 time units and E=4 time units. Notice that the requests occur in such a way that the queue will always be empty.

In Figure 15, the first request does not occur until time T, and the second and third requests occur almost immediately thereafter. Thus, from time S to time T, the queue is empty. From time T to time V, the queue length is two and the supervisor is servicing the first request. From time V to time X, the queue length is one and the supervisor is servicing the second request. From time X to time Y the queue is empty.

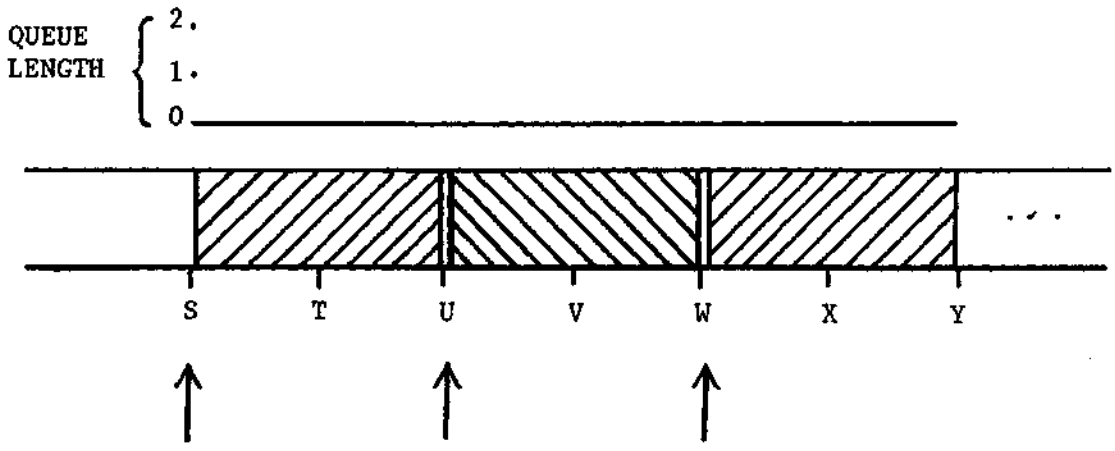


Figure 14. No Clustering, No Queueing

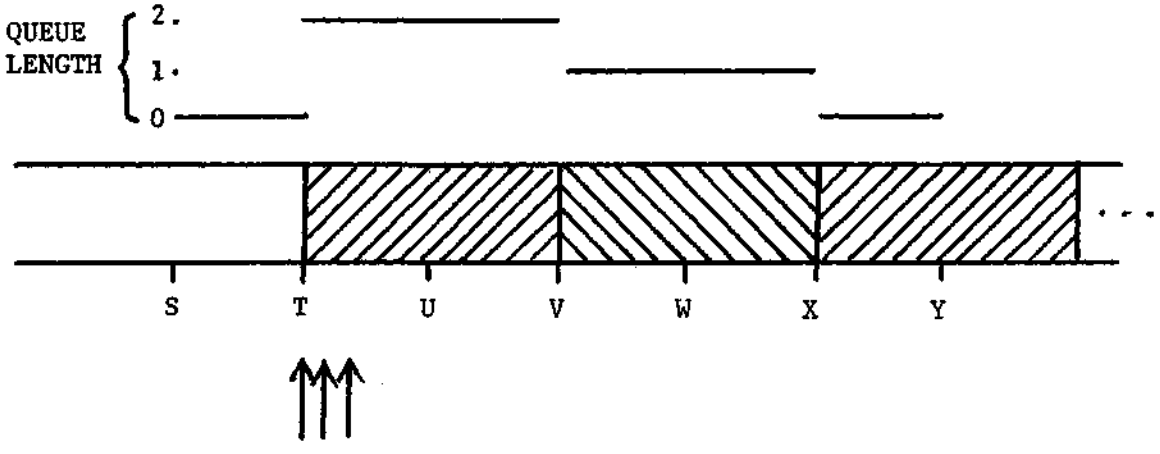


Figure 15. Queueing Caused by Clustering

Therefore, the average queue length in this case is:

$$(1x0 + 2x2 + 1x2 + 1x0)/6 = 1. \qquad (8)$$

Notice that this second example would have the same average parameter values as the first example: N=3, L=2, E=4 ((2+4+6)/3).

These examples are intended to show that the reason a queue will build up is due to the fact that sometimes requests for service occur in clusters. Requests occur in clusters because of the randomness of the interrequest time distribution. In Figure 14, the interrequest times for the three processors were all two time units. In Figure 15, the interrequest times were different: 2 time units, 4 time units, and 6 time units. While the examples are obviously extreme cases, the clustering of requests will occur in general.

3.3.4 CRS and Randomness

It has been demonstrated that the randomness in a workload is a factor which contributes to the degradation of a computer's performance. It has been stated that CRS requires a large variety of jobs in the workload. Thus, CRS needs and actually benefits from the variety which is normally detrimental. That is, CRS makes use of an "undesirable" characteristic of the workload. This point is very important, as it demonstrates the power of performance design.

These comments indicate that perhaps First-Come-First-Serve would be a superior scheduling philosophy if the workload had little variety, but CRS would be better if it had much more variety. The question of how much more variety is necessary for CRS is a difficult one, but one which can be answered for an important special case.

Consider the situation where the block size is the average supervisor service time and the maximum number of blocks which can be allocated for each request is one. At the end of any current service time, there would be less than N processors executing problem programs and thus less than N blocks of supervisor time scheduled for the service of those future requests. If there were at this time N different processor cycle lengths (measured in blocks) in the mix, then there must be at least one job which could be scheduled into the supervisor's schedule. Thus, based on the distribution of processor cycles of jobs in the mix, the mix size could be determined by requiring that at least N different jobs be available at all times. A method to accomplish this is developed below.

Assume that the block size is I, the number of jobs in the mix is M, and the processing cycles of the jobs in the mix follow the distribution function F(x). Also, if X is the largest possible processing cycle size, define K as X/I.

Table 1 exemplifies the number of jobs in the mix which have a processing cycle of length 1,2,...,K.

Define p_i as the probability that a sample, Z, from F(x) will fall in block i. Then:

$$P_i = \text{Prob}((i-1)I \leq Z \leq iI) \qquad i = 1,\ldots,K \tag{9}$$

$$P_i = F(iI) - F((i-1)I) \qquad i = 1,\ldots,K \tag{10}$$

Define P* as the probability that, of M jobs, r_1 fall in block 1, r_2 fall in block 2,...,r_K fall in block K. Using the multinomial dis-

Table 1. Mix Distribution

Number of Blocks	Number of Jobs of This Size
1	4
2	3
3	5
4	0
5	1
6	2
7	7
8	3

tribution,

$$P^* = \frac{M!}{r_1!r_2! \ldots r_K!} \quad p_1^{r_1} p_2^{r_2} \ldots p_K^{r_K} \tag{11}$$

Define P_j as the probability that j of the K blocks are non-empty, when the total number of jobs is M. Then

$$P_j = \sum_{<r_1 r_2 \ldots r_K>} P^* \quad , \tag{12}$$

where $<r_1 r_2 \ldots r_K>$ is all sets such that

$$\text{a)} \quad \sum_{i=1}^{K} r_i = M \quad ,$$

and b) at least j of the r_i are non-zero (13)

Thus, for the previous case, the proper mix size, M^+, could be determined by

$$M^* = \text{smallest M such that } P_N > \delta \tag{14}$$

where δ is the desired acceptance level, i.e., the probability that the scheduling algorithm can find a job to schedule and N is the number of processors.

$$M^+ = M^* + N + (\text{maximum number of busy channels}) + (\text{maximum supervisor queue}) + (\text{maximum channel queue}) + (\text{supervisor utilization}) \tag{15}$$

That is, M^+ is equal to the number of jobs which must be available in

the mix, M^*, plus the number of jobs busy at the various resources in the system. It should be noted that this derivation is contingent on the assumption that the mix distribution, F(x), is stable.

If the number of blocks allocated for each request is not limited to one, then the analysis becomes much more difficult. However, it is certain that a larger mix would be necessary for this case than the previous case. Before a job is scheduled in this case, it must meet two requirements: a processor cycle length that ends at an empty block and a supervisor service time that ends before the next busy supervisor block. In the previous case, only the first requirement was necessary. Thus, more variety of jobs in the mix is necessary for the case when the number of blocks allocated is unlimited. While the formulation of M^+ is not explicitly used in what follows, it does demonstrate the type of analysis which can be made for that case.

3.3.5 Amount of Improvement Through CRS

Recall that the example of section 3.1.4 demonstrated that the use of blocks introduces some queueing into the behavior of the system. Figure 16 provides a similar example. Job P_2 interrupts at the end of the block numbered two and job P_3, at the beginning of block three. Thus, job P_2 is likely to still be using the supervisor when job P_3 makes its request.

To provide an estimate of the amount of queueing introduced, consider the model in Figure 17. The arrows indicate a request for the supervisor. Assume that every block contains one and only one arrow and that the service time for every request is constant and equal to the block size. Also assume that each request arrives uniformly between the

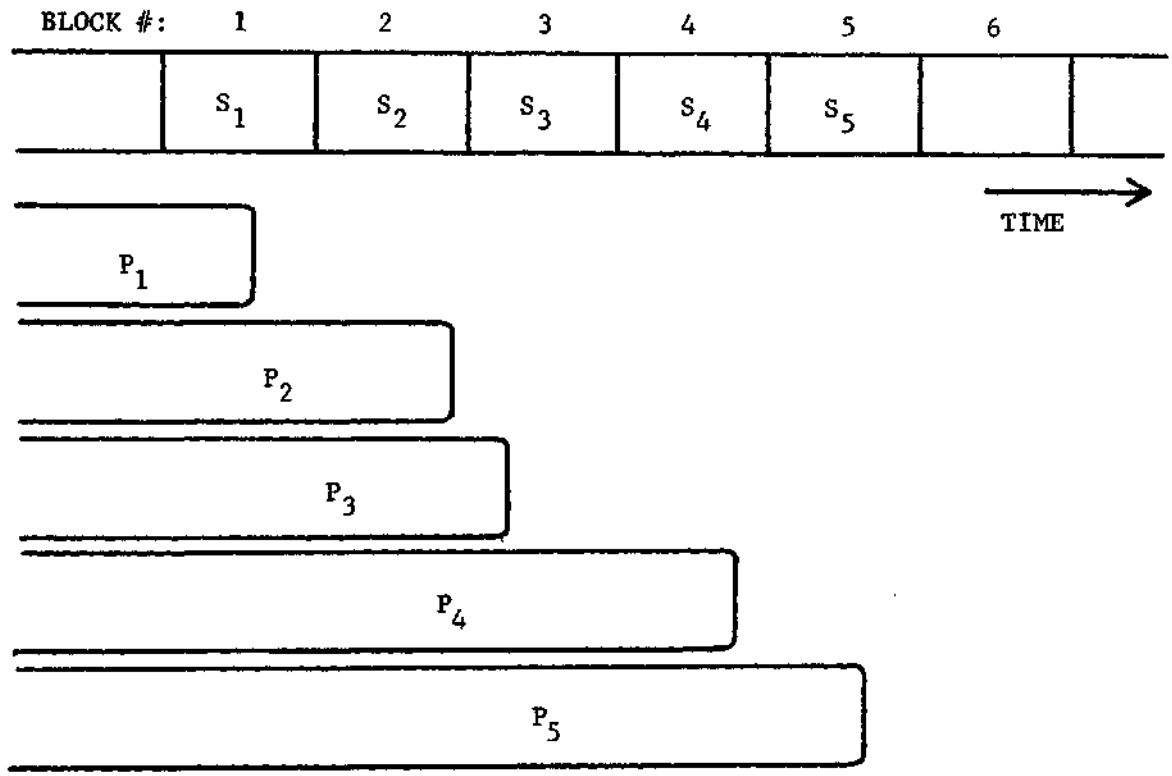


Figure 16. Example of CRS

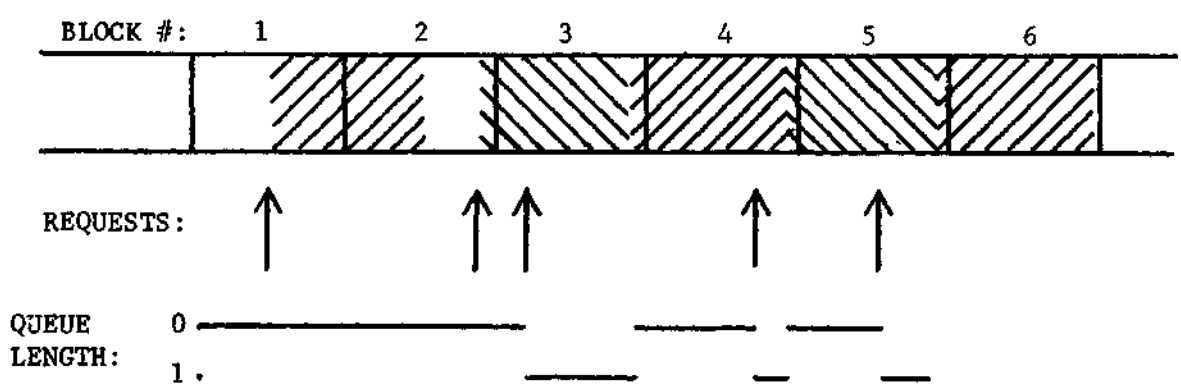


Figure 17. Corresponding Model of CRS

beginning of the block and the end of the block. Under these assumptions, the supervisor queue must always be of either length zero or length one. This is indicated in the example.

After the initial transients of blocks one and two, the system would reach equilibrium. The request which occurs in block four is actually handled in block five (i.e. from the beginning of block five to the end of block five). This means that the request which occurs in block five must wait on the queue until the beginning of block six. This is true for every block after the initial transient. On the average, a request will occur in the middle of a block. Therefore, on the average, the queue will be empty from the beginning of the block to the middle of the block and will be of length one from the middle of the block to the end of the block. This indicates that the average queue length will be 1/2.

While this simple model does provide an estimate of the queueing introduced by blocks, it is inaccurate for two reasons. First, queueing would tend to be less since a request will probably not occur in every block. Second, queueing would tend to be more since supervisor service times are not likely to be exactly constant. The effect of the interaction of these two factors is, perhaps, best studied through simulation.

3.3.6 Supervisor Load

"Load" is a queueing theory concept which can be useful in analyzing and extending the results of Madnick's model. The "offered load" is defined as the mean number of requests per service time and is denoted by the symbol "a" (30). This is essentially a measure of the demand

placed on the server. For queueing systems with one server and the restriction that requests for service wait on a queue until being served, the offered load is equal to the server utilization (30). Based on Madnick's model and equation 6, the offered load can be specified as in equation 16:

$$a = \frac{L}{L+E} (N-Q) \tag{16}$$

Notice that the Q on the right side of equation 16 is not an independent variable, but rather is dependent on L, E, and N. This reflects the fact that the offered load can not be computed directly but depends on the system to which the load is offered (30).

The "intended offered load," a^*, is defined as the load that would be offered if the system had as many servers as needed so that queueing could not occur. For our situation, this would correspond to having N processors and N supervisors. It can be shown that (30):

$$a^* = N \left[\frac{L}{L+E}\right] \tag{17}$$

The intended offered load is in general useful in this model because it provides an upper bound for the offered load. For our situation, it can be used to provide an estimate of the number of processors that would fully utilize the supervisor, for a given L/E ratio.

For example, consider the case where L = 2, E = 4, and N = 3. On the average, each of the three processors would use the supervisor for 2 time units and then would process for 4 time units. If these processors each use the supervisor for a third of the time (2/2+4),

then it would be reasonable to expect the processors together would use the supervisor all of the time. This was illustrated in Figure 14 assuming L and E were constant. If they were random, full utilization could be attained by scheduling such that all the periods of supervisor utilization exactly filled the supervisor schedule.

Based on Madnick's model, the upper bound on utilization desired would be

$$a^* = 100\% \quad . \tag{18}$$

From equations 17 and 18,

$$1 = N\left[\frac{L}{L+E}\right] \quad ; \tag{19}$$

or

$$N = 1 + \frac{E}{L} \quad . \tag{20}$$

Equation 20 provides the bound on the number of processors for full supervisor utilization under perfect scheduling. This relation could be used by system designers to determine the maximum feasible number of processors which the supervisor could support for a given L/E ratio.

This relation can be used in this study to help interpret the graphs in Figure 13 as they relate to the effectiveness of CRS. If the values of N as specified in equation 20 was plotted on Figure 13, it would be evident that this is the point on the curve at which the relation between the queue size and the number of processors becomes almost linear. This should be expected, as equation 20 was derived by con-

sidering the supervisor as being 100% utilized. Thus, if $N \leq 1 + E/L$, the entire queue, as specified in equation 4, could theoretically be eliminated through CRS. The additional queueing introduced by $N > 1 + E/L$ could not be eliminated by any scheduling algorithm, since the supervisor would already be 100% busy.

3.3.7 Implementation Considerations

Several points should be made about the actual implementation of this algorithm. First, it is expected that some of the information required by this algorithm is likely to be needed by other parts of the supervisor (54,79). For example, processing cycle lengths may be part of the system accounting data. Another factor which would tend to decrease the overhead incurred by this algorithm is that there is likely to be hardware support for the supervisor of a larger multiprocessor system (32).

The introduction of allocation by blocks makes hardware support of this algorithm particularly simple. For example, consider the special case in which the block size is the average supervisor time and only one block is allocated for each request. The supervisor schedule could be maintained as a hardware register, as illustrated in Figure 18. A "1" could indicate an available block and "0" could indicate a previously scheduled block. The current time would correspond to the left end of the register, and time could be elapsed by a simple one bit shift to the left. This register would need to be long enough to permit the scheduling of the longest jobs in the mix.

The variety of jobs in the mix could be represented in another register, as in Figure 18. A "1" in the i^{th} bit from the left would

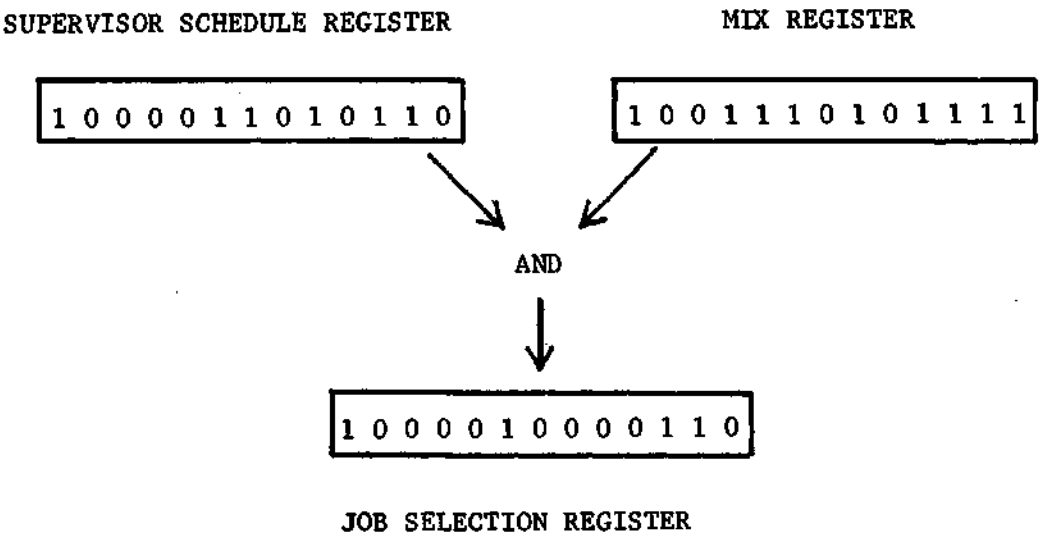


Figure 18. Hardware Support for CRS

indicate that there is at least one job in the mix which has a processing cycle i blocks long. The search for jobs which could be scheduled at the current time could then be accomplished by simply AND-ing the supervisor schedule register and the mix register into a third register, as in Figure 18. A "1" in this register would point to a job in the mix which meets the scheduling requirements. One of these could then be selected on a basis of some external criterion such as priority or on an internal basis such as scheduler optimization. A content-addressable memory could be effectively used to support this part of the search. Thus, CRS could be made as fast as much simpler, static scheduling algorithms through the intelligent use of hardware technology.

CHAPTER IV

SIMULATION RESULTS

4.1 Initial Investigation

4.1.1. Model Description and Validation

The initial investigation was intended to determine if there were any special problems or considerations which should be taken into account in the second, more detailed, model. As previously stated, my model for this investigation was based on Madnick's model of a computer system. Scheduling was via CRS with blocks.

The simulation program consisted of about 140 GPSS statements. Due to the restrictive nature of GPSS, 95% of these statements were necessary to represent CRS, with the remaining 5% representing the computer system model.

To help validate the simulation model, an initial experiment was performed with FCFS scheduling. The validation experiment demonstrated that the GPSS simulation model gave results essentially the same as that given by equation 4.

4.1.2 CRS Effectiveness

In general, the initial investigation indicated that CRS could be an effective means for eliminating supervisor queueing (47). For example, if N = 21 and L/E = .05, then FCFS scheduling would give an average supervisor queue length of 2.8. CRS gave an average queue length of 0.7 to 1.7, depending on various parameter values. These results were assuming no forecasting errors.

Intuitive expectations were generally supported by the experimental results. A decrease in the block size resulted in an increase in cost, in terms of simulation run time, and an increase in effectiveness, in terms of supervisor queueing. With the block size equal to the average supervisor time, supervisor queueing could not be reduced below an average of one half.

The development of the GPSS model provided some useful ideas concerning the implementation of CRS in the simulation model. While some of the problems resolved here would not be encountered in an actual implementation, several problems were basic to the development of the algorithm. Some of these problems could be intuitively explained, but were not so obvious that they were initially anticipated.

For example, consider the technique for maintaining a pointer which indicates the current time on the supervisor schedule. It was initially thought that this pointer could be adjusted at the end of a service interval by pointing to the end of the block which corresponded to the supervisor request just handled. This, of course, implies that a block is not simple marked as "taken," but rather is marked as "taken by job X."

Since some queueing does occur, the time at which a service interval was scheduled to complete does not always correspond to the actual time at which it completes. Therefore, at each schedule point, the pointer must be computed from the actual time by taking a clock reading modulo the block size, and taking that result modulo the number of blocks in the supervisor schedule. This, of course, assumes that the supervisor schedule is implemented as a circular list. For example, if

there were 100 blocks in the supervisor schedule and the block size was 50 time units, then a clock reading of 21475 time units would point to block: (21475 mod 50) mod 100 = 429 mod 100 = 29.

The preliminary investigation also indicated the importance of having additional tuning factors in the algorithm besides simply the block size. When more than one block can be allocated for a request, then the question arises as to whether to allocate an additional block when, for example, only an additional half of a block is actually needed. The portion of a block needed before an additional block is allocated was found to be a useful tuning factor.

For example, consider two alternative values of such a tuning factor: a) 90% and b) 25%. That is, in case (a), an additional block will be allocated only if 90% of an additional block is needed. If the block size were 100 time units and the supervisor time for a particular request were 370 time units, case (a) would allocate three blocks while case (b) would allocate four blocks. In general, the higher the percentage, the less idleness would be scheduled into the supervisor's schedule and the more queueing would be introduced. While it may at first seem that 50% would be the best value of this timing factor, it must be determined whether this would schedule too much idleness into the supervisor schedule.

4.1.3 Job Selection Bias

The initial investigation pointed out an inherent problem with CRS: Jobs with short processing cycle tend not to be selected for scheduling. It is not unusual for non-trival scheduling algorithms to be biased against jobs with some particular characteristic. The usual tech-

nique for dealing with this problem is dynamic priority assignment, i.e., the longer a job stays in the mix unscheduled, the higher the priority of that job is raised. Eventually these jobs reach a sufficiently high priority to insure that they get scheduled. While dynamic priority assignment does prevent jobs from remaining idle too long, excessive use of it could be detrimental to the primary goal of a scheduling algorithm since high priority jobs must be scheduled regardless of any performance criteria.

An investigation was made to determine the cause of the bias. The results of this investigation are explained through an example which follows.

Assume that I, M, F(x), K, and P_i are as defined in section 3.3.4. Also assume that L=I, M is large, the smallest processing cycle is one block, and the supervisor schedule could be as illustrated at the top of Figure 19. Ignoring any blocks which may have been scheduled prior to the current time, t_0, consider the next job selection. At time t_0, the selected job could request any of the blocks 1,2,...,K. Thus, by time t_0, each of the blocks 1,2,...,K have had one opportunity to be filled. This, also, is illustrated in Figure 19. Assuming the next job is scheduled at time t_1, then blocks 2,3,...,K would have had two opportunities to be filled, t_0 and t_1, and block K+1 would have had only one opportunity, t_1. Continuing this process, after time t_{K-1}, block K would have had K opportunities, block K+1, K-1 opportunities, ..., and block K+(K-1), 1 opportunity. Thus, after the initial transient of blocks 1,2,...,K-1; equilibrium is reached in which the blocks closer to the current time have had many more opportunities to be scheduled

TIME:	t_0	t_1	t_2	t_3	t_4	t_{K-1}	. . .	t_{K+3}	. . .				
BLOCK NO. OF SUPERVISOR SCH:	1	2	3	4	...	K	K+1	K+2	K+3	...	2K-1	2K	2K+1	2K+2
NUMBER OF SCHEDULE ATTEMPTS BY TIME:														
t_0	1	1	1	1	...	1								
t_1	---	2	2	2	...	2	1							
t_2	---	---	3	3	...	3	2	1						
t_3	---	---	---	4	...	4	3	2	1					
...														
t_{K-1}	---	---	---	---	...	K	K-1	K-2	K-3	...	1			
t_K	---	---	---	---	...	---	K	K-1	K-2	...	2	1		
t_{K+1}	---	---	---	---	...	---	---	K	K-1	...	3	2	1	
t_{K+2}	---	---	---	---	...	---	---	---	K	...	4	3	2	1
...														

Figure 19. Job Bias

than blocks farther away. The probability that each of these blocks have actually been filled is, of course, dependent on F(x). If, for example, F(x) is the uniform or exponential distribution, then the probability that the blocks close to the current time have been filled is much more than the probability that the blocks farther away have been filled. Thus, it is likely that jobs with short processing times will not be able to be selected. The effect of this bias will be explained in section 4.3.2.

In summary, the preliminary investigation has indicated that CRS can effectively increase system throughput by reducing supervisor queueing but has inherent shortcomings which must be dealt with in a way which will not interfere with the main purpose of CRS.

The next several sections describe the results of experimentation with the more detailed GASP simulation model.

4.2 Model Definition Experiments

Four experiments were performed which serve to analyze the GASP computer system model with respect to Madnick's computer system model and the CRS scheduler.

4.2.1 Submodel Validation

The first model definition experiment helps validate the correctness of the GASP simulation program by specifying GASP parameters such that the detailed computer system model is reduced to a form equivalent to that assumed in Madnick's analytic model and then comparing the results of the simulation with the analytic results.

One of the parameters used to accomplish this was the number of processing cycles of the jobs; this was specified as one. This, of

course, implies that there would be no I/O cycles. The scheduling algorithm was FCFS, which is the same as in Madnick's model under these conditions. The processing cycle and supervisor service distributions were both exponential. Various L/E ratios and number of processors were used, with the results given in Table 2. The fourth column of Table 2 is the queue length provided by the queueing theory model, and column five is the average queue length from four replications with the GASP simulator. The last column indicates the results of the statistical verification of this experiment. The null hypothesis that the simulation queue length has an average value equal to the queueing theory result is tested against the alternate hypothesis that they are not equal. This and later statistical tests are standard; so, the detailed computation will not be given here. The result for this experiment was that the null hypothesis was not rejected at the 95% level.

It should be noted that while the number of replications for this and other experiments is nominal, the intent of the experiments is not to prove anything, but rather to help guide the development and analysis. Due to the large number of model parameters, and possible parameter values, any attempt to prove anything for all cases would be hopeless.

4.2.2 Full Model Validation

The next experiment was similar to the first, except that simulation parameters were chosen to exemplify the full capabilities of the GASP model.

The mix size was restricted to 80 jobs. The processor cycle lengths were specified to be uniformly distributed, and the parameters

Table 2. Submodel Validation

Run	L/E	Number of Processors	Expected Queue Length: μ_0	Average Simulated Queue Length: $\bar{X}$	Reject Hypothesis H_0: $\bar{X}=\mu_0$ at 95% level
1	.050	21	2.76	2.53	No
2	.050	30	9.18	9.10	No
3	.050	15	.930	.92	No
4	.034	21	.998	1.08	No
5	.034	30	3.46	3.45	No
6	.034	15	.374	.397	No
7	.071	21	4.79	4.54	No
8	.071	30	13.01	12.26	No
9	.071	15	1.61	1.76	No

for each jobs processing distribution were generated from a uniform distribution. The mean processor cycle length was 10,000 time units. The I/O cycles followed a uniform distribution around 10,000. The I/O initiate service time was exponentially distributed with various means and the I/O complete service time was uniformly distributed with a mean of 30 time units. I/O complete interrupts were handled when they occurred by interrupting the processor which has the longest processing time remaining. The number of processing cycles of the jobs followed a uniform distribution with a mean of eleven. Various numbers of processors were used, with the number of channels equal to the number of processors.

The results of this experiment are given in Table 3. The simulated queue lengths were again close to the values predicted by the queueing theory model, but not as close as in the first experiment. The average absolute difference between $\bar{X}$ and μ_o in Table 2 was .199 and in Table 3 was .276, an increase of 38%. In spite of this increase in difference, only Run 6 was shown to be significantly different.

4.2.3 I/O Complete Technique Comparison

The third model definition experiment compared the two basic techniques for handling I/O complete interrupts that were described in section 3.2.3. As that description explained, it should be expected that throughput would be better for the case of delayed handling of I/O complete interrupts since this case does not require processors to be reassigned as often as the case where I/O complete interrupts are handled immediately. Other model parameters follow Run 1 of the previous experiment, except that CRS is used instead of FCFS. The block size

Table 3: Full Model Validation

Run	L/E	Number of Processor	Expected Queue Length: μ_0	Average Simulated Queue Length: X	Reject Null Hypothesis
1	.050	21	2.76	2.74	No
2	.050	30	9.18	8.30	No
3	.050	15	.930	1.07	No
4	.034	21	.998	1.00	No
5	.034	30	3.46	3.58	No
6	.034	15	.374	.439	Yes
7	.071	21	4.79	4.43	No
8	.071	36	13.01	12.2	No
9	.071	15	1.61	1.70	No

was equal to the average supervisor time, and there was no limit placed on the number of blocks allocated for each request. It was also assumed that the length of all processing cycles was exactly known.

The basic measure for comparison in this and later experiments was throughput--the total processing and I/O done in a specified interval, i.e., during the specified elapsed time of the simulation run. The average throughput for delayed I/O complete was $\bar{X}_2=6.625 \times 10^6$ time units during a period of 200000 time units for four replications. (The "$x10^6$" will hereafter be understood.) For non-delayed I/O complete, the average throughput was $\bar{X}_1=6.143$. To determine if the improvement gained through delaying the handling of the I/O complete was significant, the null hypothesis that the two means were equal was tested against the alternate hypothesis that they were not equal. At the 95% level, the null hypothesis was rejected, indicating that the difference was significant. Therefore, the remaining analysis will follow the delay I/O complete alternative.

4.2.4 Channel Queueing Effect

The last model definition experiment considers the effect of the number of channels in the computer configuration on CRS. The mix size is related to the average number of jobs waiting for a channel. For example, if the average length of the channel queue is five, then the number of jobs available for selection by the scheduling algorithm is five less than if the average channel queue length was zero. Thus the variety of jobs available to CRS is affected by the number of channels in the configuration.

It seems reasonable to suspect that the reduction in variety

caused by channel queueing could be partially offset by increasing the maximum number of jobs allowed in the mix by the corresponding amount. An experiment to test this theory was carried out by first making a simulation run with enough channels so that there would be no channel queueing and obtaining an average throughput, $\bar{X}_1$. In the second run, the number of channels was restricted so that there would be some queueing. Finally, a third run was made with a restricted number of channels but with the maximum number of jobs allowed in the mix increased by the size of the channel queue in run two, and it gave a throughput of $\bar{X}_2$.

With four replications, $\bar{X}_1 = 6.76$ and $\bar{X}_2 = 6.63$. The null hypothesis; H_0: $\bar{X}_1 = \bar{X}_2$. At the 95% level, the null hypothesis was not rejected. Another important factor was that the channel queue did not increase significantly from run two to run three. These results indicate that the effect of the number of channels in the configuration is just like the effect of the mix size--both affect the variety of jobs in the mix. For this reason, the interaction between the number of channels in the configuration and CRS was not directly studied, but rather was incorporated into the study of the relation between the mix size and CRS, which will be discussed later.

4.3 Algorithm Development Experiments

Eight experiments were performed which completely develop CRS based on the computer system defined in the last section. These experiments are primarily concerned with the elimination of the bias in job selection which was introduced in section 4.1.3.

4.3.1 Mix Search Technique

The first two experiments analyze the possibility of eliminating

the bias by searching through the mix in a particular order when searching for a job to meet the requirements of CRS. The example of 4.1.3 indicated that the bias against jobs with short processing cycles was caused by the fact that the periods of time close to the current time in the supervisor schedule had more opportunities to be filled than periods farther away. It seems reasonable to suspect that, through searching the mix in a particular order, it may be possible to schedule jobs such that available periods of supervisor time would exist for short jobs as well as long jobs.

Seven search algorithms were compared. Algorithm A ordered the mix according to the amount of time each job had been ready to process. The mix was then searched so that jobs which had been waiting for a processor the longest would be considered for scheduling first. This is obviously a desirable search technique since it attempts to keep jobs from remaining idle too long. Algorithms B through G ordered the mix according to the size of the next processing cycle of each job in the mix. Algorithm B searched the mix on a basis of shortest processing time first. This approach to eliminating the bias against short jobs could be interpreted as giving "first choice" of available space to short jobs. As will be indicated in the results which follow, these first two algorithms did not eliminate the problem; so, the following modifications were examined. Algorithm C searched the mix on a basis of longest processing time first. Algorithm D alternated between longest first and shortest first; that is, one time the longest job in the mix would be selected; and the next time the shortest job in the mix would be scheduled. Algorithm E alternated between longest first and shortest

first after every third scheduling of a job instead of after every scheduling. Finally, Algorithm F and G alternated after every 15th and 25th scheduling, respectively. It was initially believed that these algorithms which alternate would provide some randomness in the filling of the supervisor schedule not provided by the previous algorithms.

Most of the model parameters for this experiment were adopted from the experiments of section 4.2. However, this experiment, as well as the rest of the experiments in this section, were performed under two basic cases. Case A specifies the block size to equal the average supervisor service time and restricts the number of blocks allocated for each request to one. Case B allows the block size to vary in general, but for this experiment sets it to equal the average supervisor service time. This case also allows the number of blocks allocated for each request to vary as needed. Implicit in case A is the assumption one block of time is often enough to complete service and that an extraordinary number of blocks is seldom needed. This is reasonable because the functions associated with the handling of an I/O initiate interrupt include various bookkeeping duties plus the search for a job to schedule. Approximately the same bookkeeping duties would be performed at each I/O initiate interrupt, so this could possibly be represented by a constant distribution. The distribution of the time required to search the mix would depend on the scheduling algorithm. For CRS, the search time would vary, perhaps uniformly, between the time required to consider the first job in the mix to the time required to consider the last job in the mix.

The results for case B are given in Table 4. The second column

Table 4: Algorithm Development Experiment 1B

Algorithm	Average Throughput	No Job %
A	9.39	16.
B	9.53	21.
C	9.22	73.
D	9.64	27.
E	9.34	30.
G	9.30	37.

gives the average throughput for four replications, and the third column indicates the percent of time that CRS could not find a job which met the processing and supervisor service time requirements. When this occurred, CRS would arbitrarily select the job with the shortest processing cycle for scheduling and therefore introduce additional queueing of requests to the supervisor.

The various throughputs were compared by a one-way analysis of variance. In this (and later) experiments, the null hypothesis that the mean throughput for all treatments (algorithms) were equal was tested against the alternate hypothesis that the means were not all equal. At the 95% level, the null hypothesis was not rejected.

The "NO JOBS" percents indicate that none of the algorithms successfully eliminated the job bias: Bias against selection of a particular type of job results in more jobs of that type in the mix and less jobs of other types which results in not enough variety to meet the needs of CRS. However, some of the algorithms were obviously better than others at minimizing the degree of the bias.

The results for case A are in Table 5. Algorithm G was not considered in this case. The NO JOB statistics followed the same pattern as in Table 4, but were numerically smaller. The cause of this reduction was that the jobs selected in case B must meet two requirements (processor cycle length and supervisor service time) while those selected in case A need meet only one (processor cycle length).

However, the bias against short jobs was present for every algorithm in case A. For example, the distribution of the processing cycle lengths (in terms of number of blocks) for algorithm A after 200,000 time

Table 5. Algorithm Development Experiment 1A

Algorithm	Average Throughput	No Job %
A	6.76	0
B	6.75	0
C	6.41	36.
D	6.64	13.
E	6.67	12.
F	6.54	17.

units of simulation is given in Table 6. Another indication of the presence of the bias was the histogram of the average length of time that a job of a specified processing cycle length waited in the mix ready to process. For algorithm A again, jobs which had a processing cycle length of one block waited an average of 81000 time units in a ready state, those of length two waited 50000 time units, those of length three waited 43000 time units, those of length four waited 38000 time units, those of length five waited 34000 time units, etc. The average waiting time was 39000; so, short jobs had to wait longer than the average.

4.3.2 Prescheduling

These results indicated that a different approach to eliminating the bias would be necessary. It has been previously stated that a dynamic priority allocation scheme could be used, but would probably degrade the performance of the system. This expectation was verified through simulation, but the details of that experiment will not be given here. Prior to describing the methodology developed to solve the bias problem, an example of the problem will be given to motivate the approach.

Consider the example distribution of jobs in the mix given in Table 7. Assume that jobs entering the mix have processing cycles uniformly distributed between 1 and 20 blocks long but that the current mix distribution has more small jobs due to the bias. In the example, the number of available jobs is forty. Thus, if there were no bias, there should be twenty different jobs, with two of each size. But due to the bias, there are only ten different jobs--those of length 1,2,3,4,5,6,7,

Table 6. Mix Distribution for Algorithm A

Number of Blocks	Number of Jobs of This Size	Number of Blocks	Number of Jobs of This Size
1	8	21	1
2	6	22	0
3	3	23	1
4	2	24	1
5	0	25	1
6	2	76	1
7	1	27	2
8	2	28	0
9	3	29	1
10	2	30	1
11	1	31	3
12	0	32	4
13	0	33	1
14	1	34	1
15	4	35	1
16	2	36	1
17	3	33	0
18	1	38	0
19	2	39	3
20	1	40	0

Table 7. Example Mix Distribution

Number of Blocks	Number of Jobs of This Size
1	12
2	7
3	7
4	4
5	1
6	3
7	1
8	3
9	0
10	1
11	0
12	0
13	0
14	1
15	0
16	0
17	0
18	0
19	0
20	0

8,10, and 14. Therefore the bias problem not only provides poor performance, in terms of turnaround time, for small jobs, but also is detrimental to the performance of CRS since it reduces the variety of the jobs in the mix.

If one of the jobs one block long could be scheduled, then two desirable effects would result. First, one of the small jobs which had waiting in the mix for a long time would get processed. Second, the variety of the jobs in the mix would be increased if the next processing cycle of that job was of length 9,11,12,13,15, etc.

To accomplish this, define τ as a limit on the number of the same size jobs allowed in the mix at the same time. For example, a value of three may be used for τ for the case in Table 7. Thus, the expected number of like jobs would be two, and the maximum number of like jobs would be three.

There would be two situations when a job would need to be added to the distribution of available jobs: when a job first entered the mix and after an I/O complete interrupt. When either of these two situations occurred, then the number of jobs already in the mix of the same size as the job to be added would be compared to τ. If this number was less than τ, then the job would simply be added to the mix. If this number was equal to τ, then the job would be added to the mix and a "prescheduling" algorithm would be invoked.

This algorithm would remove that one of the $\tau+1$ jobs of the same size as the new job which had been in the mix the longest. This job would then be prescheduled; that is, it would be scheduled to be scheduled some time in the future and the associated supervisor blocks would

be allocated.

For example, consider the situation illustrated in Figure 20. The current time is at the end of the block prior to block one, and the S's indicate the blocks previously allocated to jobs i, n, j, and m. Suppose that at the current time a job of size three was added to the mix and there were already τ jobs of that size in the mix. The one of those that had been in the mix the longest, job k, would be selected for prescheduling. Assume that job k requires two blocks of supervisor time. If job k was to be normally scheduled, then it would be placed into a processor and blocks four and five would be allocated. However, there is no processor available and blocks four and five are not available. So instead, blocks farther down the supervisor schedule are allocated. Figure 21 illustrates that blocks ten and eleven could be selected. The distance of the allocated blocks to the current time should be random, but must be longer than the processing cycle length. If the I/O initiate for job k is to occur in block ten, and the job processes for three blocks, then job k must start processing at the end of the interrupt handled in block six. Thus, at the end of the handling of an interrupt and prior to the normal scheduling procedure, CRS would need to decide if a prescheduled job needs to be scheduled. An easy way to accomplish this would be to maintain a preschedule array as illustrated in Figure 21. When job k was allocated blocks ten and eleven, a "k" would be placed in block six of the preschedule array. At the end of the handling of the interrupt for job j, CRS would determine that it must schedule job k.

Thus the prescheduling procedure can force biased jobs to be

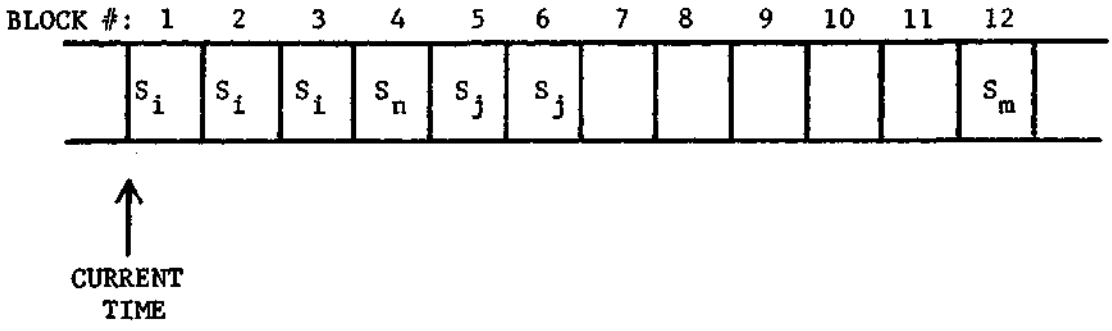


Figure 20. Normal Supervisor Schedule

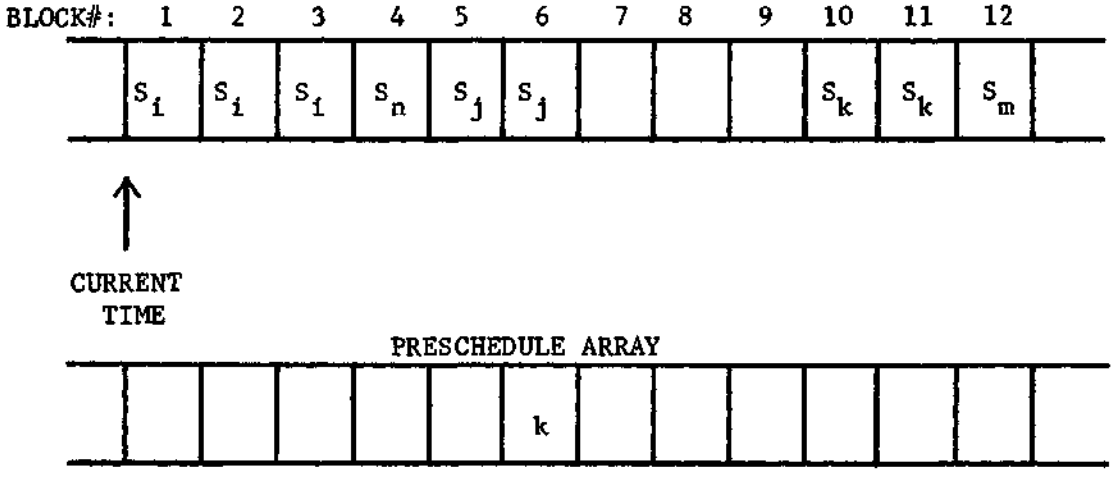


Figure 21. Prescheduling of Job k

executed without degrading the performance of CRS. In fact, it could possibly improve the performance since a maximum variety of jobs in the mix is maintained.

A question which must be considered with regard to this procedure is the extent to which the procedure should be applied. Prescheduling can be thought of as a means of reserving space in the supervisor schedule for jobs which otherwise would have difficulty finding space. If prescheduling is used too extensively, most of the supervisor's schedule could be reserved, leaving no room for normal scheduling.

The second pair of algorithm development experiments analyze this problem by comparing various threshold values. The results for case A are given in Table 8.

The maximum variety of jobs, as specified by the processing cycle distribution, was 40, and the mix size was 100. So, on the average if 20 jobs were elsewhere in the system, there should have been two available jobs of each size in the mix. The tested values for the maximum number of jobs of each size allowed in the mix were 2, 4, 5, 6, and 8. The average throughput for four replications were compared by a one-way analysis of variance as before, and the null hypothesis that all means were equal was not rejected at the 95% level. Thus, if prescheduling eliminated job bias, it did so without degrading the performance of the algorithm.

The fourth column of Table 8 indicates the average percent of the jobs scheduled which had been prescheduled. The 68% for run 1 was found to be too high for the reason suspected: The supervisor schedule was so full of reserved spaces that not enough room was available for normal

Table 8. Prescheduling: Case A

Run	Threshold	Average Throughput	Percent of Schedules Prescheduled
1	2	6.66	67.7
2	4	6.76	10.9
3	5	6.78	4.7
4	6	6.75	1.2
5	8	6.76	0.

scheduling. This was evidenced by a NO JOB percent of 3.1 and by examination of the supervisor schedule during simulation. The NO JOB percents for all other alternatives were zero. The threshold for run 5 was too high, since prescheduling was never performed. The intermediate threshold in run 3 resulted in a reasonable prescheduling percent of 4.7 and a slightly higher average throughput than the other alternatives. For this case, the histogram of the average waiting time versus the processing cycle size verified the elimination of the bias. Jobs one block long waited an average of 40000 time units, those two blocks long, 42000 time units, those three blocks long, 42000 time units, those four blocks long, 47000 time units, those five blocks long, 41000 time units, those six blocks long, 34000 time units, etc. The average waiting time for all sizes was 40000 time units. The remaining case A experiments reported in this chapter maintain a prescheduling percent of about 5.

The results for case B are given in Table 9. Just as in case A, the null hypothesis that the mean throughput for all alternatives were the same was not rejected at the 95% level. The prescheduling percentage followed the same pattern as in case A. A threshold of six provided a reasonable percentage of 4.0.

For case B, the percentage of time that no job could be found to meet the CRS requirements was higher at every threshold value than for algorithm A with no prescheduling. This indicated that, for case B, prescheduling did more harm than good. After further investigation, this was found actually to be true.

To help explain the cause of this phenomenon, consider the example which was given in Figure 21. Recall that job k was prescheduled

Table 9. Prescheduling: Case B

Run	Threshold	Average Throughput	Percent of Schedules Prescheduled	NO JOB %
1	2	9.13	45.6	28.6
2	3	9.34	31.2	23.4
3	4	9.22	14.9	20.2
4	6	9.35	4.0	16.0
5	8	9.32	.79	17.2

for blocks 10 and 11 and had a processing cycle length of three blocks. Suppose, instead, that the processing time for job k was four blocks. The k for the preschedule array would have then been placed in block five. But no supervisor service interval ends in block five, so job k could not have been scheduled at the proper time.

An attempt was made to alleviate this problem by removing any job which was unsuccessfully prescheduled and re-prescheduling that job for a later time. However, since most of the preschedule jobs were small, it was discovered that often the blocks freed by removing the prescheduled job from the supervisor schedule would never be refilled.

Thus, for case B, the prescheduling of jobs reserved space which was never used. This not only reduced the amount of available space in the supervisor schedule, but also artifically scheduled idleness for the supervisor.

So prescheduling, as defined here, was found to be a very effective means of dealing with the job bias problem for case A, but counter-productive for case B.

4.3.3 Mix Size Comparison

Up to this point, a mix size of 100 has been used. The following pair of experiments analyzes the effects of the mix size on CRS.

Some of the current single and dual processor systems allow very large mixes, sometimes more than a hundred jobs. It is doubtful that a system with 21 processors could maintain a hundred jobs for each processor. The mix sizes compared here are in the more reasonable range of from four to twelve jobs for each processor.

Any scheduling algorithm does not depend on the mix size itself,

but rather on the number of jobs in the mix available for scheduling. CRS depends on the number of different jobs in the mix available for scheduling. As previously mentioned, the number of jobs available is affected by the I/O boundness of the jobs and the number of channels available. This study assumes there are enough channels to handle most requests. If this were not the case for any particular system, then the proper mix size for that system would have to be adjusted according to the average channel queue length to obtain a number of available jobs equivalent to that associated with the mix size found to be appropriate by this study.

The results for case A are given in Table 10. The null hypothesis that the throughputs were the same was not rejected at the 99% level, indicating that all of the mix sizes tested were of sufficient size to support CRS.

Column 4 gives the average number of different jobs available. The average, for every case, is above the 21 needed to insure a job is available to meet the requirements of CRS, as described in section 3.3.4. The measured standard deviation for run 1 was 1.8. Assuming a normal distribution, this would indicate that a mix size of 84 would provide at least 25 different jobs in the mix 98% of the time.

The increment in mix size from run 1 to run 2 provided for a much larger increase in the number of different jobs than the increment from run 2 to run 3. To further insure that a job could be found for scheduling, advantage was taken of this nonlinear relationship by concluding that a mix size of 100 seemed appropriate for this case. This size mix resulted in a 98% probability that at least 28 different jobs

Table 10. Mix Size: Case A

Run	Mix Size	Average Throughput	Average Number of Different Jobs
1	84	6.73	28.3
2	126	6.83	55.4
3	168	7.01	38.3
4	210	7.03	39.3

would be available in the mix.

Table 11 summarizes the results for case B. Just as in case A, the null hypothesis that all throughputs were equal was not rejected. The throughput figures here were adjusted because the larger mix sizes required longer simulation before reaching equilibrium.

The standard deviations of the number of different jobs varied from 3.4 for run 1 to 4.4 for run 5. Also the average number of different jobs for a given mix size was lower for case B than case A. These two results, combined with the fact that case B requires more variety than case A as explained in 3.3.4, indicate that a larger mix is necessary for case B.

Table 11 indicates that the difference in the average number of different jobs for each increase in the mix size becomes small after a mix size of 168 jobs is reached. Assuming a normal distribution as before, associated with this mix size is a 98% probability that there will be at least 24 different jobs in the mix. No mix size larger than this substantially increased the variety of jobs in the mix or substantially decreased the NO JOB percentage. So, for lack of a better alternative, it was concluded that a mix size of 168 was the minimal necessary for this case.

4.3.4 Tuning

As indicated in section 4.1.2, a decision must be made as to how much of an additional block is needed before one additional block is allocated. Of course, this only applies to the case where more than one block can be allocated. The effectiveness of this as a tuning parameter will now be analyzed through an experiment which considers

Table 11. Mix Size: Case B

Run	Mix Size	Average Throughput	Average Number of Different Jobs
1	84	31.28	24.0
2	126	31.21	29.7
3	168	31.16	31.9
4	210	30.99	33.4
5	252	31.41	33.6

several alternatives.

The second column of Table 12 indicates the percent of a block needed before an additional block is allocated. For example, if the block size is 100, and a service interval is 260, then runs 1, 2, and 3 would allocate three blocks while runs 4 and 5 would allocate only two blocks. The restriction is made that at least one block is always allocated.

The null hypothesis that all throughputs were equal was rejected at the 95% level, indicating that this was a useful tool for tuning the system. Run 4 provided for a 4.4% increase in throughput over run 1. Column four indicates the corresponding difference in supervisor queue length caused by the difference in the amount of idleness scheduled into the supervisor schedule.

4.3.5 Block Size

Another parameter which can be modified for case B is the size of the block. Recall that the use of this parameter as a basis for a cost-effective tradeoff was the primary motivation for the introduction of blocks. It was initially believed that as the size of the block got smaller, CRS would become more accurate and therefore more effective. Of course, the accuracy obtained in this way would be limited by the accuracy of the forecasting of processor cycle lengths to some extent.

Table 13 gives the results of a comparison between the use of the block size equal to the average supervisor time, run 1, and a much smaller size, run 2. The size of the block in run 2 was chosen as small enough to represent the effects of small blocks but large enough to allow the simulator to fit into the Burroughs' memory. The average

Table 12. Tuning

Run	Percent of Block Needed Before Allocated	Average Throughput	Average Supervisor Queue
1	20	13.99	2.58
2	40	14.15	2.57
3	60	14.22	2.07
4	80	14.61	1.73
5	100	14.49	1.85

Table 13. Block Size

Run	Block Size	Average Throughput	Average Supervisor Utilization
1	500	15.01	82.4%
2	60	15.79	83.6%

throughputs for four replications of each run were compared by testing the null hypothesis that they were equal against the alternate hypothesis that they were not equal. The null hypothesis was rejected at the 95% level, indicating that the use of the smaller block size improved throughput by about 5%. However, the average supervisor utilization did not show a corresponding increase. The null hypothesis that they were equal was not rejected at the 95% level. These two contradictory results prompted further investigation into the scheduling process for small block sizes. It was noticed that there was still a bias in the selection of jobs with respect to the size of the processing cycle. But this was not the cause of the contradictory results. It was discovered that there was another type of bias. There was a tendency to select jobs which had short supervisor service times. This caused processors to spend less time in the supervisor state and more time in the problem program state; this resulted in less use of the supervisor and more problem program throughput. Thus, the 5% increase in throughput for the smaller block size was the result of this bias.

4.3.6 Summary of Algorithm Development Experiments

One consistent theme has pervaded the results of the algorithm development experiments as well as the preliminary considerations of section 3.3. The case where the number of blocks allocated for each request was limited to one provided superior results in every instance as compared to the case when no limit was made. In section 3.3, estimates were made of the mix size and the quantity of queueing introduced by blocking for case A, but no estimates could easily be made for case B. Hardware support for case B would be more complex than that for

case A as described in section 3.3.7. The comparison of the mix search alternatives indicated that the bias was more pronounced in case B, and it was also shown that it could not be efficiently eliminated through prescheduling. Case B required at least a 50% larger mix than case A, and the reduction of the block size for case B could not be done without introducing another bias.

All of these facts pointed toward the same conclusion: CRS with case B was not a feasible approach. The problems associated with case B could possibly be reduced through considerable extension of the methodology, but the complexity introduced would probably be prohibitive.

Therefore, it was concluded that CRS would be feasible only with prescheduling and with only one block allocated for each request.

This, then, completely specifies CRS. The next section will determine the effectiveness of CRS by comparing it to a standard scheduling algorithm.

4.4 Evaluation of CRS

4.4.1 Effect of Forecasting Error

During the developmental experiments, it was assumed that the algorithm had exact knowledge of all processing cycle lengths. However, before the true advantage of CRS over standard algorithms can be determined, the error in the forcasts of processing cycle lengths must be included in the model.

Forecasting errors were assumed to be normally distributed with a mean of zero. The mean throughputs for various standard deviations of

error were compared, with the results in Table 14. The null hypothesis that the average throughputs from four replications were all equal was rejected at the 95% level, indicating that forecasting error does cause a significant degradation in the performance of CRS.

The average simulated throughput from four replications for FCFS scheduling was 6.16. Based on this information, column four of Table 14 indicates the percent increase in throughput over FCFS for CRS at each level of forecasting error. From this it is evident that the advantage of CRS is so large that even fairly substantial forecasting errors do not degrade the effectiveness of CRS below a worthwhile quantity.

4.4.2 Estimation of Actual Forecasting Error

The obvious question at this point would be which run of Table 14 corresponds to a realistic error distribution. An indication of the answer to this question was obtained, as well as an indication of the complexity of the forecasting technique necessary.

The Univac 1108 on campus was used to obtain a sequence of processing and I/O cycles for a small set of programs. Several programs were selected which were available for modification. Included were a FORTRAN production program, a COBOL production program, and several other randomly selected FORTRAN programs.

All of these programs but one had to be initially modified to run on the Univac. Then two subroutines were added which read the Univac's interval clock. One subroutine read the processor-time clock, and the other read the core-time clock. Calls to these subroutines were placed before and after each I/O instruction so that the processor-time clock

Table 14. Forecasting Errors

Run	Standard Deviation of Error Distribution	Average Throughput	Percent Increase in Throughput Over FCFS
1	0	6.78	10.04
2	5%	6.73	9.24
3	10%	6.66	8.10
4	15%	6.57	6.64
5	20%	6.57	6.64
6	35%	6.53	5.99
7	50%	6.48	5.18

could measure processing period lengths and the core-time clock could measure I/O period lengths.

To test this method of monitoring, a test program was written which did the same computation during each processing period and output the same thing during the I/O period. This test provided information which was used to refine the procedure.

The test indicated that the monitored information should be maintained internally in core until the program's normal end, and then should be written out. This reduced the effect of monitoring to adding a very small amount of processing time to each processing period. It was also determined that it would be necessary to run in real time mode, with unused peripherals made inactive, and with no one else using the machine. (This test also located a hardware error in one of the internal clocks, which was corrected by the field engineer.)

The complex procedure just described severely restricted the number of programs which could be monitored. This was not considered to be a serious problem, as the intent was to obtain some indication of actual forecasting errors, and not to study job characteristics or forecasting as an end in itself.

After the monitored data was obtained, it was modified to take any buffering into account, as described in section 2.5.2. Then three forecasting schemes were applied and distributions of the percentage error in forecasts were measured for the three forecasting schemes under various parameter values on all monitored programs with two buffer sizes.

The first forecasting routine was the standard double moving

average on a linear model. It is assumed that the sample at time t, x_t, is a linear function of time:

$$x_t = a_1 + a_2 t + \epsilon_t \quad . \tag{21}$$

The forcasting routine based on this model is

$$\hat{x}_\theta = \hat{b}_1 + \hat{b}_2 \theta \quad , \tag{22}$$

where

$$\theta = t - \frac{N+1}{2} \tag{23}$$

In (22), $\hat{x}_\theta$ is the estimate of the sample and $\hat{b}_1$, $\hat{b}_2$ are estimates of the intercept and slope for the time origin change of (23). Values for $\hat{b}_1$ and $\hat{b}_2$ are computed recursively as a moving average over the last N samples:

$$\hat{b}_1 = \hat{b}_1 + \frac{x_i - x_{i-N}}{N} \quad ; \tag{24}$$

$$\hat{b}_2 = \hat{b}_2 + \frac{12}{N(N^2-1)} \left(\frac{N-1}{2} x_i + \frac{N+1}{2} x_{i-N} - N\hat{b}_1\right) \quad . \tag{25}$$

A complete derivation is available in (15).

The second forcasting routine was a dynamic double exponential smoothing formula, as developed in (79). The forecast, $\hat{x}_t$, is defined by:

$$x'_t = \alpha x_{t-1} + (1-\alpha) x'_{t-1} \quad ;$$

$$x''_t = x'_t - x'_{t-1} \quad ;$$

$$x'''_t = \beta x''_t + (1-\beta)x'''_{t-1} \quad ;$$

$$\hat{x}_t = x'_t + x'''_t \quad . \tag{26}$$

It should be noted that $\hat{x}_t$ consists of two components: x'_t and x'''_t. Here, x'_t is an exponential estimator of the average of the samples and x'''_t is an exponential estimator of the trend in the samples.

Normally, the exponential smoothing parameters, α and β are maintained as constants with a value between zero and one. The methodology described in (79), and adopted in this study, allows these parameters to vary according to previous errors in prediction as described below.

The percentage error in the previous forecast is defined as

$$e_{t-1} = \frac{\hat{x}_{t-1} - x_{t-1}}{x_{t-1}} \quad . \tag{27}$$

If e_{t-1} if larger than some acceptable threshold, τ, then the signs of e_{t-1} and e_{t-2} are compared. If the signs are the same, then it is assumed that the forecasting routine is not responsive enough and so α and β are increased so that recent history is taken more into account as described below. If the signs are different, then it is assumed that the forecasts are oscillating and so α and β are decreased so that the forecaster relies more on historical samples.

The modification is made by first determining if α is at an extreme (either 0 or 1, whichever is appropriate). If it is not, then

α is modified by some small amount d_α. If it is, and if β is not already at its extreme, then β is modified by d_β. Otherwise, no modification is made.

The third forecasting routine was developed through discussions with Donovan Young specifically for the processing cycle time series. It seems reasonable to expect that many programs execute in a loop for some number of interactions, where there is one or more I/O statements in that loop, and the number of instructions executed during each pass through the loop is approximately the same. Then control would likely transfer to a different loop with a different number of instructions. Such a pattern of execution is illustrated in Figure 22. A forecasting scheme for this pattern should use an average within a program loop but also react rapidly to changes from one program loop to another.

To accomplish this, the standard exponential smoothing formula was slightly modified:

$$\hat{x}_t = \begin{cases} \alpha x_{t-1} + (1-\alpha)\hat{x}_{t-1} & \text{if } \left|\frac{x_{t-1} - \hat{x}_{t-1}}{x_{t-1}}\right| < \tau \\ x_{t-1} & \text{otherwise.} \end{cases} \qquad (28)$$

The forecast, $\hat{x}_t$, is based on exponential smoothing if the error in the previous forecast was less than some threshold τ. Otherwise, the forecase is equal to the previous sample value. Thus, when a change from one loop to another is encountered, the exponential average is updated to the new level.

It is obvious that both the double exponential and linear forecasters require more computation than the modified exponential smoothing.

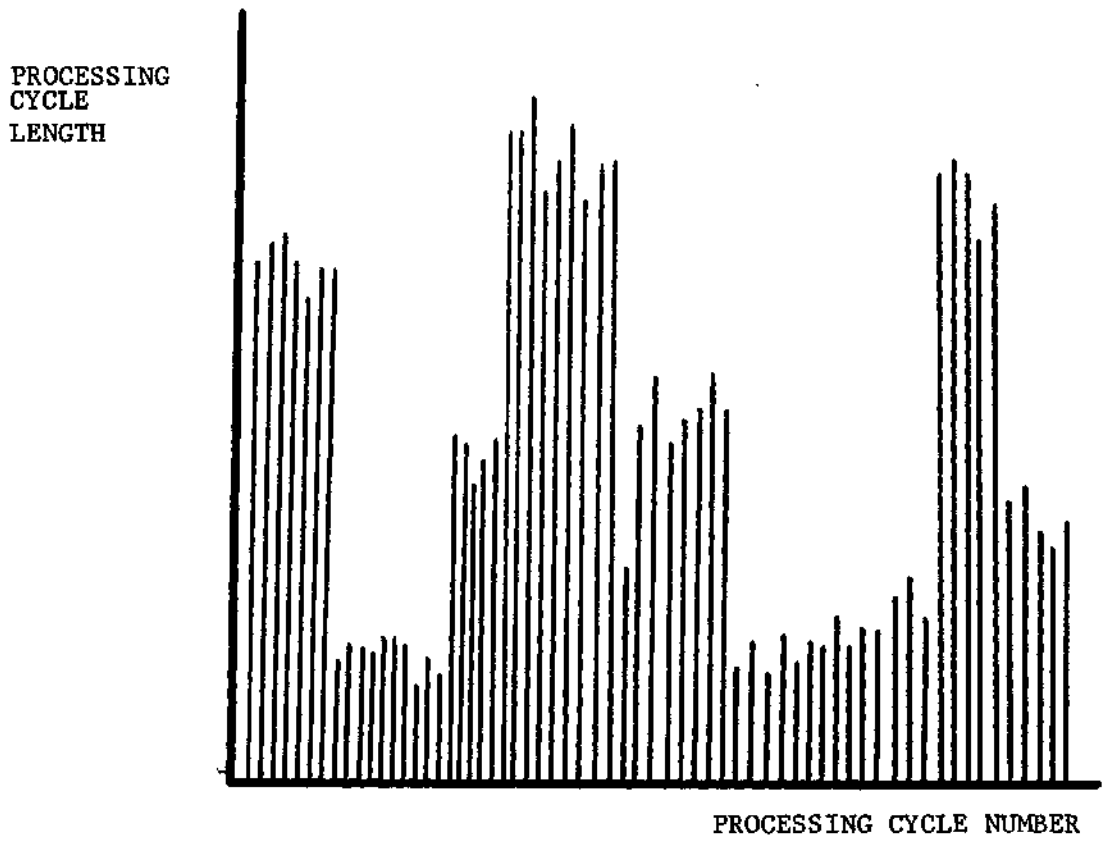


Figure 22. Processing Cycle Time Series

Also, the data base necessary to support modified exponential smoothing is much less than the other two. The linear method requires N+2 words of storage for every program in the mix: the N previous samples and previous values of $\hat{b}_1$ and $\hat{b}_2$. Double exponential smoothing requires four words for each program in the mix: α; β; x'_{t-1}; x''_{t-1}. Modified exponential smoothing requires only one: $\hat{x}_{t-1}$.

Table 15 gives a summary of the results of this investigation. The measure of accuracy in the second column is the percent of the forecasts, averaged over nine time series, which fall between -15 and +15 percent error.

The double moving average routine gave the worst results. Values of N between five and eleven were tested. The best results occurred for values around seven.

Double dynamic exponential smoothing was only slightly better. The parameter τ was varied between .05 and .3, but little effect was noticed.

The best forecasting scheme was the one designed specifically for the type of time series involved. Every combination of values of α and τ from .1 to .9, .1 apart, were tested. Best results were obtained for α = .1 and τ = .6. This was reasonable because the low α makes past history important within a loop and τ = .6 detects a change from one loop to another when the error exceeds 50%.

The special case of the modified exponential smoothing when $\tau = 0$ is the simplest forecasting scheme:

$$\hat{x}_t = x_{t-1} \quad . \tag{29}$$

Table 15. Forecast Routine Comparison

Forecasting Routine	Average Percent of Forecasts within ±15% Error
Double Moving Average	43.0
Double Exponential Smoothing	44.5
Modified Exponential Smoothing	74.4
Modified Exponential Smoothing with $\tau=0$	62.5

The fourth row of Table 15 indicates that this simple technique performed very well. This supports the idea that processing cycle lengths are somewhat repetitive in nature.

The general shape of the distribution of error was similar to that of the normal distribution, as should be expected. For the normal distribution, 68% of the sample fall within $\pm\sigma$, where σ is the standard deviation. Based on the programs monitored, it would be reasonable to estimate that 68% of the forecasts fall within $\pm$ 15% error. Thus, the error distribution should be normal with a mean of zero and a standard deviation, σ, of 15%.

This estimate corresponds to run four of Table 14. Therefore it would be reasonable to conclude that the use of CRS could increase throughput by 6.6% for the situation defined by the parameters used in this analysis.

This result, coupled with the speed of the algorithm from hardware support and the simplicity of the forecasting scheme applied at the end of every processing cycle, indicates that CRS could provide a significant increase in the amount of work done for a large system with little increase in cost over conventional techniques.

4.4.3 Load Conditions

The investigation to this point has been for one set of values for Madnick's parameters: L/E = .05, 21 processors. As indicated in section 3.3.6, this corresponds to a load condition of high supervisor utilization. It is assumed that the effectiveness of CRS under these parameter values is equivalent to its effectiveness under other parameter values which correspond to high supervisor utilization; i.e.,

L/E = .10, 11 processors; L/E = .02, 51 processors; etc.

However, it may be that CRS would perform even better under less loaded conditions. This might be expected since CRS would be able to spread requests out more; that is, the block size used could be larger than the average supervisor interval, providing more leeway for errors introduced by blocking and forecasting.

To determine the validity of this argument, first a comparison was made between CRS and FCFS under different load conditions, with the results in Table 16. The number of processors was varied from 19 to 17 to 15 while L/E remained at .05. FCFS was compared to CRS with a forecasting error distribution with a standard deviation of 15% through the null hypothesis that the average throughputs were equal. For each run, the null hypothesis was rejected at the 95% level. Column five of Table 16 gives the percentage increase in throughput provided by CRS.

A comparison of the effectiveness of CRS under various load conditions is illustrated in Table 17. The number of processors and percentage improvement of CRS over FCFS were taken from Tables 14 and 16. The fourth column gives the percent increase in the number of processors available to process when all of the supervisor queueing, as determined from the queueing theory model, is eliminated. The last column indicates the percentage of possible improvement attained by CRS for the various load conditions. Thus CRS could perform as much as 50% better under less loaded conditions.

4.4.4 Note on Experiments

The experiments described in this chapter certainly do not ex-

Table 16. Load Conditions

Run	Number of Processors	CRS Average Throughput	FCFS Average Throughput	Percent Improvement Of CRS over FCFS
1	19	6.16	5.73	7.46
2	17	5.59	5.26	6.27
3	15	4.98	4.75	4.63

Table 17. Relative Effectiveness

Run	Number of Processors	Percent Improvement of CRS over FCFS	Percent Improvement Predicted by Queueing Theory Model	Percent of Possible Improvement Attained
1	21	6.64	16.0	41.5
2	19	7.46	12.2	62.2
3	17	6.27	9.36	65.3
4	15	4.63	7.11	65.1

haust all possible considerations and alternatives which could be investigated, and they were not meant to do that. Rather, these experiments were chosen because they seemed to provide the basic information necessary for this investigation while not requiring an excessive amount of computer time. The experiments described here required over 80 hours of CPU time on the Burroughs B5700, and the program monitoring required over 12 hours of exclusive use of the Univac 1108.

CHAPTER V

CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

5.1.1 General Remarks

This research has developed a feasible methodology for improving throughput in large multiprocessor systems by reducing the amount of queueing of requests to the supervisor. The methodology was based on developments in the areas of multiprocessor configuration design, operating system design, scheduling, workload characterization, and forecasting of workload requirements. The use of the combination of these areas to solve this problem in system design was guided by the concept of performance design.

5.1.2 Summary of Results

The problem was identified. The supervisor of a multiprocessing system could be considered to be a resource which can be requested by processors. If more than one processor request this resource at a time, a queue of requests would develop, causing idle processors and a corresponding reduction in throughput.

Then performance design was defined as the incorporation of performance evaluation techniques into the system design such that the system could dynamically respond to the immediate requirements of the workload in order to improve system performance. This was adopted as the basis for a methodology to solve the problem.

A model of a computer system was developed to be realistic

enough to enable relevant considerations to be studied. It was based on a multiprocessor, multichannel, multi-memory module structure with maximum communication among components. The GASP II simulation language was used to implement the model as a 2800 statement FORTRAN program.

From this model and the standard scheduling algorithms, it was determined that a 21 processor system with a reasonable supervisor load would have an average of 2.6 processors idle on the supervisor queue and would provide an average throughput of 6.16.

To improve throughput, a dynamic scheduling algorithm was developed to schedule jobs to processors such that they would request the use of the supervisor when no other job was predicted to be using the supervisor. This capability was based on the assumed knowledge of the sequence of intra-interrupt intervals for each job in the mix.

During the development of the algorithm, the effects of many factors deemed relevant were fully investigated. Inherent problems in the methodology were solved through experiments which prompted the development of a "look-ahead" technique.

Using this scheduling algorithm with the assumption that job characteristics were exactly known, throughput was increased by 10% to 6.78 over the standard scheduling algorithms. Since it is not likely that this information would be exactly known, the effect of introducing various amounts of error were investigated (see Table 14). Then a realistic estimate of expected error was obtained by applying several forecasting techniques to data obtained through software monitoring of actual programs. A forecast of the next intra-interrupt period was

based on the lengths of the previous intra-interrupt periods. A modified exponential smoothing formula performed best, providing an error distribution that was normally distributed with a zero mean and a standard deviation of about 15%. This amount of error reduced throughput to 6.57, 6.6% above standard scheduling techniques.

Analysis of different configurations indicated that the dynamic scheduling algorithm would perform even better than the example used here if the supervisor was not as heavily utilized.

5.2 Extension of CRS

5.2.1 Abstraction of Essential Features of CRS

While the development of CRS in this thesis has been with respect to a particular application in multiprocessor system design, the basic idea behind CRS is general in nature and could conceivably be applied to other problem areas in computer system design. As a first step in the extension of CRS to other areas of application, a description of the algorithm will be given in terms independent of any particular application.

Assume a system is composed of at least three basic types of resources: the "clustered" resource, the "scheduled" resource, and the "held" resource. Consider the essential activity of the system as being described by the statement that a "requesting unit" utilizes some portion of the scheduled resource and then utilizes all of the clustered resource while, at the same time, also utilizing part of the held resource. It is also assumed that more requesting units sometimes leave the scheduled resource and request the clustered resource than the clustered resource can "service." That is, a queue of requesting

units may form at the clustered resource. An important point is that while on the clustered resource queue, a requesting unit does not actually need the use of the held resource, but is unable to release it. Thus, a requesting unit on the clustered resource queue is preventing other requesting units from having access to the held resource. Since there is a cost associated with the operation of the held resource, there is a cost associated with the portion of the held resource which is inactive because of requesting units being on the clustered resource queue. This directly imples that there is a cost associated with the waiting of a requesting unit on the clustered resource queue.

The object of CRS is to reduce the cost associated with this queue. The method is to schedule requesting units to the scheduled resource such that they request the clustered resource when no other unit is requesting that resource. CRS is based on the premise that the length of time a requesting unit uses the scheduled resource and clustered resource is known, at least to some degree of accuracy.

5.2.2 Areas of Application

For the multiprocessor design problem considered in this thesis the clustered resource is the supervisor, and both the scheduled resource and the held resource correspond to the processors.

As an example of a different interpretation, consider the problem of paging. CRS could possibly be applied by scheduling jobs to processors such that they request the use of the paging drum at a time when no other job is requesting it. In this case, the drum is the clustered resource, the processors correspond to the scheduled resource, and

memory is the held resource. While there may be some problems in the application CRS to paging, this example at least demonstrates that the idea behind CRS is a general one, and could possibly be extended to other resource allocation problems.

The further development of other application areas is considered a worthwhile area for future work.

REFERENCES

1 Amiot, L., Natarajan, J.K., and Aschenbrenner, R.A. "Evaluating a Remote Batch Processing System," Computer, v5, #5, (1972), p24.

2 Arbuckle, R.A. "Computer Analysis and Thruput Evaluation," Computers and Automation, v15, #1, (1966), p12.

3 Arndt, R.F. and Oliver, G.M. "Hardware Monitoring of Real-Time Computer System Performance," Computer, v5, #4, (1972), p25.

4 Aschenbrenner, R., Amiot, L., and Natarajan, N.K. "The Neurotron Monitor System," AFIPS FJCC, v39, (1971), p31.

5 Baer, J.L. "A survey of Some Theoretical Aspects of Multiprocessing," ACM Computing Surveys, v5, #1, (1973), p31.

6 Bell, C.G. and Newell, A. Computer Structures: Readings and Examples. McGraw-Hill: New York, (1971).

7 Bell, C.G., Chen, R., and Rege, S. "Effect of Technology on Near Term Computer Structures," Computer, v5, #2, (1972), p29.

8 Bhandarkar, D.P. Analytic Models for Memory Interference in Multiprocessor Computer Systems Ph.D Dissertation Carnegie-Mellon (1973).

9 Boehm, B.W. "Computer Systems Analysis Methodology," RAND Corp. Document #R-520-NASA, (1970).

10 Bookman, P.G., Brotman, B.A., and Schmitt, K.L. "Use Measurement Engineering for Better System Performance," Computer Decisions, v4, #4, (1972), p28.

11 Bordsen, D.T. "Univac 1108 Hardware Instrumentation System," Proc. ACM First SIGOPS Workshop on System Performance Evaluation, (1971), p1.

12 Box, G.E.P. and Jenkins, G.M. Time Series Analysis Forecasting and Control, Holden-Day: San Francisco, (1970).

13 Bremer, R.W. and Ellison, A.L. "Software Instrumentation Systems for Optimal Performance," Proc. IFIP Congress, v1, (1968), p520.

14 Bright, H.S. "A Philco Multiprocessing System," AFIPS SJCC, v18, (1964), p97.

15 Brown, R.G. Smoothing, Forecasting, and Prediction of Discrete Time Series. Prentice Hall: London, (1962).

16 Burnett, G.J. and Coffman E.G. "A Combinatorial Problem Related to Interleaved Memory Systems," JACM, v20, #1, (1973), p39.

17 Buzen, J. "Analysis of System Bottlenecks Using a Queueing Network Model," Proc. ACM First SIGOPS Workshop on System Performance Evaluation, (1971), p82.

18 Calingaert, P. "System Performance Evaluation: Survey and Appraisal," CACM, v10, #1, (1967), p12.

19 Campbell, D. J. and Heffner, W.J. "Measurement and Analysis of Large Operating Systems During System Development," AFIPS FJCC, v33, (1968), p903.

20 Cantrell, H.N. and Ellison, A.L. "Multiprogramming System Performance Measurement and Analysis," AFIPS SJCC, v32, (1968), p213.

21 Chang, W. "A Queueing Model for a Single Case of Time Sharing," IBM Systems Journal, v5, #2, (1966), p115.

22 Chang, W., Paternot, Y.T., and Ray, J.A. "Throughput Analysis of Computer Systems--Multiprogramming versus Mutliprocessing," Proc. ACM First SIGOPS Workshop on System Performance Evaluation, (1971), p59.

23 Cheng, P.S. "Trace-Driven System Modeling," IBM Systems Journal, v8, #4, (1969), p280.

24 Clancy, J.J. and Fineberg, M.S. "Digital Simulation Languages: A Critique and a Guide," AFIPS FJCC, v27, (1965), p23.

25 Clayton, B.E., Dorff, E.K., and Fagen, R.E. "An Operating System and Programming System for the CDC 6600," AFIPS SJCC, (1964), p41.

26 Coffman, E.G. and Wood, R.C. "Interarrival Statistics for Time-Sharing Systems." CACM, v9, #7, (1966), p500.

27 Coffman, E.G. "Studying Multiprogramming Systems," DATAMATION, v13,#6, (1967), p47.

28 Control Data 6600 Computer System Reference Manual #450 (1963) Control Data Corporation.

29 Conway, R.W., Maxwell, W.L., and Miller, L.W. Theory of Scheduling. Addison-Wesley: Reading, Mass., (1967).

30 Cooper, R.B. Introduction to Queueing Theory, The Macmillan Co: New York, (1972).

31 Cooperman, J.A., Lynch, H.W., and Tetzlaff, W.H. "SPG: An Effective Use of Performance and Usage Data," Computer, v5, #5, (1972), p20.

32 Critchlow, A.J. "Generalized Multiprocessing and Multiprogramming Systems," AFIPS FJCC, v24, (1963), p109.

33 Devereauz. J.A. "An Application-Oriented Multiprocessor System--Control Program Feature," IBM Systems Journal, v6, #2, (1967), p95.

34 Drummond, M.E. "A Perspective on System Performance Evaluation," IBM Systems Journal, v8, #4, (1969), p252.

35 Emshoff, J.R. and Sisson, R.L. Design and Use of Computer Simulation Models, Macmillan Co: New York, (1970).

36 Fife, D.W. "An Optimization Model for Time-Sharing," AFIPS SJCC, v28, (1966).

37 First Annual Symposium on Measurement and Evaluation, Sponsored by ACM Special Interest Group on Measurement and Evaluation, (1973).

38 Flores, I. "Multiplicity in Computer Systems," Computers and Automation, v15, #7, (1966), p19.

39 Flynn, M.J. and Podvin, A. "Shared Resource Multiprocessing," Computer, v5, #2, (1972), p20.

40 Flynn, M.J. "Some Computer Organizations and Their Effectiveness," IEEE Transactions on Computers, vC21, #9, (1972), p948.

41 Flynn, M.J. "Shared Internal Resources in a Multiprocessor," Proc IFIP Congress 71, v1, (1972), p565.

42 Foster, C.C. "A View of Computer Architecture," CACM, v15, #7, (1972), p557.

43 Fox, D. and Kessler, J.L. "Experiments in Software Modeling," AFIPS FJCC, v31, (1967), p429.

44 Fuchs, E. and Jackson, P. "Estimates of Distributions of Random Variables for Certain Computer Communication Traffic," CACM, v13, #12, (1970), p752.

45 Glinka, L.R., Brush, R.M., and Ungar, A.J. "Design, Thru Simulation, of a Multiple-Access Information System," AFIPS FJCC, v31, (1967), p437.

46 Goutanis, R.J. and Viss, N.L. "A Method of Processor Selection for Interrupt Handling in a Multiprocessor System," Proc. of the IEEE, v54, #12, (1966), p1812.

47 Gwynn, J.M. and Raynor, R.J. "Scheduling in a Multiprocessor Environment," Proc. 1973 Sagamore Computer Conference on Parallel Processing, IEEE Cat #73 CH0812-8C, (1973).

48 Herman, D.J. and Ihrer, F.C. "The Use of a Computer to Evaluate Computers," AFIPS SJCC, v25, (1964), p383.

49 Hillier, F.S. and Lieberman, G.J. Introduction to Operations Research, Holden-Day: San Francisco (1967).

50 Holtwich, G.M. "Designing a Commercial Performance Measurement System," Proc. ACM First SIGOPS Workshop on System Performance Evaluation, (1971), p29.

51 Huesmann, L.R. and Goldberg, R.P. "Evaluating Computer Systems Through Simulation," Computer Journal, v10, #2, (1967), p150.

52 Hutchinson, G.K. and Maguire, J.N. "Computer Systems Design and Analysis Through Simulation," AFIPS FJCC, v27, (1965), p161.

53 Hutchinson, G.K. "Some Problems in the Simulation of Multiprocessor Computer Systems," Simulation Programming Languages, North-Holland: Amsterdam, (1968), p305.

54 IBM Reference Manual: OS/VS2 Planning Guide for Release 2 Form GC28-0667. IBM Corporation (1973).

55 Ihrer, F.C. "Computer Performance Projected Through Simulation," Computers and Automation, v16, #4, (1967), p22.

56 Johnson, R.R. "Needed: A Measure for Measure," DATAMATION, v16, #17, (1970), p22.

57 Katz, J.H. "An Experimental Model of System/360," CACM, v10, #11, (1967), p694.

58 Katz, J.H. "Simulation of a Multiprocessor Computer System," AFIPS SJCC, v28, (1966), p127.

59 Keefe, D.D. "Hierarchial Control Programs for Systems Evaluation," IBM Systems Journal, v7, #2, (1968), p123.

60 Kimbleton, S.R. and Moore, C.G. "A Probabilistic Framework for System Performance," Proc. ACM First SIGOPS Worksop on System Performance Evaluation, (1971), p337.

61 King, W.F. "Analysis of Demand Paging Algorithms" Proc. IFIP Congress 71, v1, (1972), p485.

62 Kleinrock, L. "Sequential Processing Machines (SPM) Analyzed With a Queueing Theory Model," JACM, v13, #2, (1966), p179.

63 Lewis, P.A.W, Goodman, A.S., and Miller, J.M. "A Pseudo-Random Number Generator for the System/360." IBM Systems Journal, v8, #2, (1969).

64 Lucas, H.C. "Performance Evaluation and Monitoring," ACM Computing Surveys, v3, #3, (1971).

65 McKinney, J.M. "A Survey of Analytical Time-Sharing Models," ACM Computing Surveys, v1, #2, (1969).

66 Madnick, S.E. "Multi-Processor Software Lockout," Proc. ACM National Conference, (1968), p19.

67 Merikallio, R.A. and Holland, F.C. "Simulation Design of a Multiprocessing System," AFIPS FJCC, v33, (1968), p1399.

68 Miller, E.F. "Bibliography on Techniques of Computer Performance Analysis," Computer, v5, #5, (1972), p39.

69 Morganstein, S.J., Winograd, J., and Herman, R. "SIM/61: A Simulation Measurement Tool for a Time-Shared, Demand Paging Operating System," Proc. ACM First SIGOPS Workshop on System Performance Evaluation, (1971), p142.

70 Nakamura, G. "A Feedback Queueing Model for an Interactive Computer System," AFIPS FJCC, v39, (1971), p57.

71 Nelson, G.W. "OPTS-600--On-Line Peripheral Test System," AFIPS FJCC, v33, (1968), p45.

72 Nielson, N.R. "The Simulation of Time Sharing Systems," CACM, v10, #7, (1967), p397.

73 Nielson, N.R. "An Approach to the Simulation of a Time-Sharing System," AFIPS FJCC, v31, (1967), p419.

74 Nielson, N.R. "ECSS: An Extendable Computer System Simulator," RAND Corp. Document # RM-6132-NASA, (1970).

75 Noe, J.D. "A Petri Net Model of the CDC 6400," Proc. ACM First SIGOPS Workshop on System Performance Evaluation, (1971), p362.

76 Northouse, R.A. and Fu, K.S. "Dynamic Scheduling of Large Digital Computer Systems Using Adaptive Control and Clustering Techniques" IEEE Transactions on Systems, Man, and Cybernetics, v SMC-3, #3, (1973), p225.

77 Oden, P. H. and Shedler, G.S. "A Model of Memory Contention in a Paging Machine," CACM, v15, #8, (1972), p761.

78 Pariser, J.J. "Multiprocessing with Floating Executive Control," IEEE International Convention Record, (1965), p266.

79 Pass, E.M. An Adaptive Microscheduler for a Multiprogrammed Computer System, Ph. D. Dissertation, Georgia Institute of Technology, Atlanta, Georgia, (1973).

80 Pritsker, A. and Kiviat, P. Simulation With GASP II, Prentice Hall: New Jersey, (1969).

81 Regis, R. "Modeling Generalized Parallel Computer Systems," Computer Research Report #14, The Johns Hopkins University, Baltimore, Maryland, (May 1971).

82 Rehmann, S.L. and Gangwere, S.G. "A Simulation Study of Resource Management in a Time-Sharing System," AFIPS FJCC, v33, (1968), p1411.

83 Sapiro, S. "A Technique to Control Waiting Time in a Queue," IBM Systems Journal, v4, #1, (1965), p53.

84 Sayers, A.P., ed. Operating Systems Survey, Auerback: N.Y. (1971).

85 Seaman, P.H. and Soucy, R. C. "Simulating Operating Systems," IBM Systems Journal, v8, #4, (1969), p264.

86 Sedgewick, R., Stone, R., and McDonald, J.W. "SPY--A Program to Monitor OS/360," AFIPS FJCC, v37, (1970), p119.

87 Shemer, J.E. and Robertson, J.B. "Instrumentation of Time-Shared Systems," Computer, v5, #4, (1972), p39.

88 Sherlock, J.F. "The Simulation of a Multicomputer System," IEEE Transactions on Computers, vC19, #1, (1970), p1114.

89 Sherman, S., Baskett, F., and Browne, J.C. "Trace Driven Modeling and Analysis of CPU Scheduling in a Multiprogramming System," Proc. First ACM SIGOPS Workshop on System Performance Evaluation, (1971), p173.

90 Smith, J.L. "An Analysis of Time-Sharing Computer Systems Using Markov Models," AFIPS SJCC, v28, (1966), p87.

91 Stanley, W. I, and Hertel, H.F. "Statistics Gathering and Simulation for the Apollo Real-Time Operating System," IBM Systems Journal, v7, #2, (1968), p85.

92 Stanley, W.I. "Measurement of System Operational Statistics," IBM Systems Journal, v8, #4, (1969), p299.

93 Univac 1100 Series: Operating System Programmer Reference, UP4144, Sperry Rand Corp.

94 Univac Reference Manual: STAT-PACK, UP4041, Sperry Rand Corp.

95 Univac 1100 Series: General Purpose Systems Simulator, UP7883, Sperry Rand Corp.

96 Watson, R.W. Timesharing System Design Concepts, McGraw Hill: New York, (1970).

97 Williams, T. "Computer Systems Measurement and Evaluation," The Computer Bulletin, (Feb. 1972), p100.

98 Witt, B.I. "M65MP: An Experiment in OS/360 Multiprocessing," Proc. 1968 ACM National Conference, (1968), p691.

99 Wuff, et al. Hydra: The Kernel of a Multiprocessor Operating System, Carnegie-Mellon Technical Report (1973).

VITA

Randy Jay Raynor was born in Raleigh, North Carolina on March 14, 1949. He graduated from Cary High School in Cary, North Carolina and received a Bachelor of Science degree in Computer Science from North Carolina State University at Raleigh in 1971. While at N. C. State, he worked with IBM at Research Triangle Park, North Carolina.

The School of Information and Computer Science at Georgia Institute of Technology granted Mr. Raynor a Master of Science degree in 1973 and a Doctor of Philosophy degree in 1974. He was appointed to positions of graduate research and teaching assistant during his graduate study.

Mr. Raynor has co-authored papers in various aspects of simulation and is a member of the Association for Computing Machinery, ACM SIGMETRICS, Phi Kappa Phi, Phi Eta Sigma, Upsilon Pi Epsilon, and Pi Mu Epsilon.